Vic Pack

Software 64

AUDIOGENIC FORTH

INSTRUCTION MANUAL

Audiogenic LTD

P.O. Box 88, Reading, Berks.

AUDIOGENIC FORTH

by Dave Middleton

The Manual

Audiogenic Forth comes in three versions, two of which are for the Commodore VIC-20, and the other for the Commodore 64. This manual covers all three versions, with most of the information being applicable to all versions, and any differences between the versions being stated.

* * * * IMPORTANT NOTE * * * *

Before inserting or removing your cartridge, please make sure that the power to the computer is switched OFF. Do NOT switch the power back on until you are sure that you have fitted the cartridge correctly. If in doubt, consult your dealer.

Installation - VIC FORTH

As we said above, the VIC FORTH comes in two cartridge versions....

1 - The low-cost version contains the FORTH program, but no extra memory built in. This version must be used in a motherboard in conjunction with an 8K or 16K RAM pack. It will NOT work if plugged straight into the cartridge slot of the VIC!

2 - The more expensive version contains the program plus 3K extra memory, occupying the usual $0400 to $1000 area in the memory map. You can use this version either by plugging the cartridge directly into the cartridge slot of the VIC, or by having it in a motherboard along with an 8K or 16K RAM pack.

Installation - FORTH 64

As the 64 contains more than enough memory for most purposes, we do not have these complications. The FORTH 64 cartridge plugs directly into the cartridge slot of the 64.

Power-Up - All versions

Once you are sure that the cartridge is fitted correctly, you can switch the power on. All three versions of Audiogenic FORTH have auto-boot, i.e. as soon as you switch on the machine, the program will be there on your screen without you having to give any RUN or SYS command. The opening screen should display the title, "Audiogenic Forth by Dave Middleton". If you do not get this message, then switch off the computer and check your installation. Every cartridge is checked before it leaves our factory, so it is unlikely that the cartridge will be at fault. However, if you cannot get your FORTH to power up correctly, consult your dealer.

USING THIS MANUAL

The best way, from my experience, to learn anything about computers is to work with the machine, trying out ideas and example programs. FORTH is ideal in this respect, there is no waiting around while editors, compilers, linkers and loaders are all shuffled between the disk and the computer. To add a new word to FORTH's vocabulary simply define it and press return! If you make an error FORTH will tell you straight away.

The intention of the manual is to give details of all the various aspects of the language without being specifically aimed at training, so it is a good idea to read through it quickly to get an insight into the language and then try and work through the example FORTH programs given in the manual. At the back of the manual are a number of tables which detail every word in the basic FORTH system, the words are grouped under logical headings for easy reference. The first table can be used for ready reference as it gives the names of all the words in the dictionary when the system is powered on.

Most of the sections are divided into three parts; a general description of specific parts of the FORTH system such as maths or stack operations, a small section on hints for using some aspect of FORTH, and a technical discussion of the operation of the language in that area. If you have never used FORTH before it would be wise to skip over the technical sections until a reasonable number of FORTH words have been learnt as no description of individual words is usually given.

This manual is not meant to be a beginner's guide to FORTH, and prior knowledge is assumed. One of the best text books I have ever read is called STARTING FORTH by LEO BRODIE and published by Prentice-Hall. It is this book as much as anything else which will make FORTH into an accepted language amongst computer programmers.

CONVENTIONS USED IN THIS MANUAL

1. All FORTH words are shown in UPPER case. When using the language FORTH expects all input to be made in lower case, except between string input/output. Using upper case in the text is only to aid in clarifiying FORTH words from normal English.

2. When an example is being given the computer's output is shown underlined.

3. All FORTH words are surrounded by 2 spaces when they occur in the main body of text eg.

.... after the IF and up to ELSE are ....

This product is protected by copyright and may not be reproduced in part or whole without the written permission of David Middleton.

(c) Copyright 1982 David Middleton and Audiogenic Ltd.

THE DICTIONARY

The dictionary in FORTH is fundamental to its operation, everything you do will make some use of the dictionary. The dictionary, like any normal dictionary, contains words and with each word comes its meaning or definition in terms of other words which will also be in the dictionary. To FORTH a word is any string of characters terminated by a space.

Unlike other languages FORTH has no reserved words, though in general it is not good practice to redefine important words as this can lead to confusion when the code is looked at in the future.

Any sequence of ASCII characters up to a maximum length of 31 characters can be combined to form the word's name. The following are all legal:-

```
<#  KEY  EMIT  */MOD  LOOP  41-ACACIA-AVE.
```

A new word is added to the end of the existing dictionary by using a colon definition, this takes the form:-

```
: STACK-POSITION   SP@ .  ;
```

The above example will create a new word in the dictionary called STACK-POSITION . The next two words have both been defined in the dictionary already SP@ puts the current value of the stack pointer onto the top of the stack. The FORTH word . (it's called 'dot') will print the top value on the stack. If the words after the name cannot be found in the dictionary, FORTH will respond with the message "xxxx say what?" where xxxx is the word that can not be found. If the word being defined already exists in the dictionary the message "xxxx isn't unique!" will be printed, compilation will continue and the word will become part of the dictionary; it is just a warning that the word is already in the dictionary. All future references to the word will be to the latest version of it in the dictionary but it should be noted that all references prior to the new definition still point to the old version.

The semi-colon at the end of the definition tells FORTH that the word has been completed. Assuming the compiler has managed to get this far the word will be made available to the user where it can be included in later definitions. If compilation fails for some reason the word will not be found in the dictionary if it is entered later.

FORTH can not guess the meaning of a word, you always have to start from the simple definitions and work up to the more complex words which will call them.

Vocabularies

So far we have seen that FORTH is a linked list of definitions which is searched to find words typed by the programmer. However there can be more than one linked list which allows words to have meaning depending on the context of the vocabulary.

For example the assembler has a word called MODE which determines the address type of an instruction. This word could also be used in a graphics package to indicate say high or low resolution plotting. If the assembler and the graphics packages are in different vocabularies then there will be no conflict between the two words. The word then has a meaning specific to its application.

The editor normally has its own vocabulary but because of the way the editor has been implemented in this system this has been removed and the editor is part of FORTH.

Example for creating a new vocabulary:-

```
VOCABULARY GRAPHICS IMMEDIATE
GRAPHICS DEFINITIONS
```

The first line creates a new vocabulary called GRAPHICS . The word is made IMMEDIATE by historical precedent. The second line prepares GRAPHICS for accepting new definitions. To stop adding words to GRAPHICS simply give another vocabulary name such as:-

```
FORTH DEFINITIONS
```

This will add any new words in to the FORTH vocabulary.

Hints....

Keep word definitions short as they are far easier to debug and read. There is no great speed advantage in writing one long definition, the JSR-RTS structure is very fast. A new word only takes a maximum of 7 extra bytes in the dictionary.

Getting Technical

For information on how words are compiled into the dictionary you are refered to the section 'How the compiler works'; this is not recommended for beginners.

MATHS IN FORTH

The maths system used in FORTH is called postfix or RPN (reverse polish notation) rather than infix or algebraic notation which we are brought up with.

Why use RPN?
What is the result from the following example?

```
12 + 10 * 3 / 5 + 2
```

It all depends on the precedence of the operators (ie. which will be calculated first). In Commodore BASIC the answer would be 20. Algebraic notation involves a lot of work in scanning for operators and maintaining temporary values. In BASIC this is not too bad but when a compiler is used a great deal of its size is taken up with expression parsing, converting the algebraic expressions into RPN!

If RPN is used sensibly it is as easy to understand as algebraic expressions. In RPN the operator follows the values. Eg.

```
10 20 +
```

10 and 20 are pushed to the stack, plus adds the top two stack values together and leaves the result on the stack, thus RPN is the obvious choice for a stack oriented language.

```
10 20 30 40 + - *
```

this is the same as 10*(20-40+30) in algebraic notation. Notice how the operators fit back into the expression.
Here is what happens:-

10 20 30 40 are pushed to the stack.
+ takes the top two values, adds them together and leaves the result.
The stack now contains:-
10 20 70
- subtracts 70 from 20 leaving -50 on the top of the stack.
The stack now contains:-
10 -50
* multiplies the top two values, leaving -500 on top of the stack.

The secret of using RPN initially is to keep it simple. Stringing together a lot of operators leads to errors.

Numbers

FORTH is an integer language working with 16 bit values, termed single precision. These can have an unsigned value between 0 and 65535 and a signed range between -32768 and 32767. The word . (dot) has already been introduced, this prints the top 16 bit value as a signed number. Another word called u. (u-dot) prints the top value as an unsigned number in the range 0 - 65536.

If a number is preceded by a minus sign it is naturally given a negative value.

```
-100 . -100 ok
       -------
```

If a number is entered containing a decimal point this will be converted to a double-precision number of 32 bits, with the range -2,147,483,648 to 2,147,483,647. A word called DPL is set to the decimal point location. If the number being entered does not contain a decimal point then DPL is set to -1.

The main reason for having double precision numbers is for the multiply, U* , and divide, U/ , routines.

Multiply takes two 16 bit values from the stack and leaves a 32 bit result. However the word * (multiply) is normally used, this drops the high 16 bits of the 32 bit number after the multiplication has taken place.

Divide requires a 32 bit number being divided by a 16 bit value and leaves a 16 bit remainder and a 16 bit quotient. Normally the word / (divide) is used, this converts a 16 bit number into a 32 bit value before the division takes place and leaves only the quotient on the stack, dropping the remainder.

Very few applications require double-precision numbers and it is probably best to avoid them until such an application comes along. Routines for multiplying and dividing double precision numbers are not provided but there are a host of routines for mixed 16 and 32 bit maths, see Table 2.

Number conversion

FORTH can operate in any number base, a very useful feature. When FORTH starts from cold the base is set to decimal but entering HEX would change it to base 16; if binary input/output is required then 2 BASE ! would be used; to restore base 10 the word DECIMAL is provided.

One of the popular BASIC programs is hex-decimal conversion. In FORTH this is very simple:-

```
HEX  12  DECIMAL  .  18 ok
                     -----

DECIMAL  15  HEX  .  f ok
                     ----
```

Literals

Any number which has not been defined as being part of the dictionary is treated as a literal (the numbers -1, Ø, 1, 2, 3 form part of the dictionary as they are used so often). If the number being compiled is less than 255 it is treated as a single byte and compiles down to 5 bytes in total:-

```
LDA #35
JSR PUSHØA
```

This is about as fast as FORTH can run, as PUSHØA is one of the most primitive words, it is not part of the vocabulary as the processor accumulator is required.
If the number is greater than 255 it is stored as a literal, a total of 5 bytes:-

```
JSR LITERAL
.WOR 1ØØØ
```

LITERAL takes the next two bytes and pushes them to the stack. The main reason for using the PUSHØA is the speed with which numbers can be moved to the stack from memory.

Hints...

In programs where speed of execution is essential avoid the use of multiply and divide routines where possible. While these routines are between 4 and 1Ø times faster than BASIC using floating point they can still cause a FORTH program to slow down considerably.

The words 2* and 2/ multiply or divide a 16 bit value by 2. Because this is performed by register rotation it is far faster than even the 16 bit addition routine!

Getting Technical

A 32 bit number is formed from two 16 bit numbers, take the following example:-

```
1Ø 2Ø M* . . Ø 2ØØ ok
             --------

1Ø 2Ø M* D. 2ØØ ok
            ------
```

Thus the high 16 bits of the 32 bit number are on the top of the stack.

Negative numbers are converted to 2's complement thus:-

```
-1 . -1 ok
     -----

-1 U. 65535 ok
      --------
```

THE SCREEN EDITOR

The editor provided with this FORTH system is very powerful and combats two of the main deficiencies in low cost microcomputers, these being the small screen size and the absence of a disk system.

FORTH is organised around 'screens' each screen being 64 characters wide by 16 lines deep, giving a total of 1Ø24 characters. In disk based FORTH systems screens are swapped transparently between the disk and RAM buffers using a virtual memory system.

With a cassette based system this is not possible being both too slow and sequential in nature. Thus it is necessary to keep the screens in RAM and save them away at convienient times. However, there is now another problem, small computers have only limited RAM. If a serious application is going to be undertaken than quite a few screens are required. To cater for this a powerful screen compression routine has been implemented. The routine compresses all spaces to between 1 and 3 bytes (a totally empty screen compresses to 5 bytes including the screen number!).

The small screen size is overcome by having a scrolling screen. The screen scrolls to the right when the cursor is within 5 characters of the right hand side of the screen. The current position of the cursor is shown on the bottom line, as well as the screen number.

Screen Ø is the interpreter screen. When return is pressed the line is passed directly to the interpreter and hence gives accesses to the rest of the language. On screens 1-255 when return is pressed the cursor just returns to the start of the next line.

Because the interpreter is only available on screen Ø it is impractical to have line editing commands defined as FORTH words, so they are available as direct key strokes. See Table 3.

Using the editor on screen Ø may seem strange for a while because FORTH's responses are not permanent. When return is pressed the whole line the cursor rests on is transferred to the TIB (terminal input buffer) where it is interpreted. Anything FORTH prints during this period does not overwrite the contents of screen Ø. When FORTH has finished interpreting a line it leaves the display on the screen, when another key is pressed (usually a cursor key) this output is erased. When FORTH has displayed data and is waiting for you to continue the cursor position is blank. ie.

S L C

For information on using the Editor please read see table 3.

Hints....

Screen Ø is very powerful and to a certain extent permanent. Only the commands DELETE , CLRØ or a system failure will erase it.

It is best to use line Ø to access other screens. Pressing home and return makes the screen immediately available. On the Audiogenic FORTH system, there is a re-edit key (F8) which jumps straight back to the screen just edited.

Use line 1 for the SSAVE (save-source) command, having it immediately accessible means you will use it more frequently.

Use line 15 for repeating a command without having to use the cursor keys to reaccess it. The cursor will remain on line 15 when return is pressed.

FORTH is very easy to crash, especially in the early stages of developing an application and for this reason the Restore key has been implemented. There are three types of action which can be implemented when restore is pressed:-

Ø RESTORE

When this is given the restore key has no effect on the system. It is set to this value when the system is powered on and thus protects applications from accidental restoration.

1 RESTORE

When the restore key is pressed now an ABORT is performed, this resets all the stack pointers and returns control to the user via Screen Ø. Both the source code and the code compiled into the dictionary are unaffected.

2 RESTORE

When developing some programs dictionary manipulation is involved. If it is performed wrongly the dictionary becomes corrupted and FORTH will crash. Pressing restore now causes FORTH to perform SET-FORTH ABORT which first sets the dictionary pointer to its cold start value and then clears the stacks.

If neither of the above work then FORTH's operating parameters have been too badly corrupted and the only action is to switch the system off and then on again.

Another way in which FORTH can be crashed is by stack overflow. When the command SP@ is given the current position of the stack pointer is left on the top of the stack. The top of the stack is at 138 and the bottom of the stack is at 18 with the stack pointer decrementing from high memory to low whenever a new value is put onto the stack. It is the stack bottom which will cause most of the problems. The word ?STACK can be used in an application to test for stack underflow or overflow and will give an error message accordingly. But if the stack has overflowed by the time ?STACK is found the system may have crashed. The way out of this is to move the bottom of the stack up so that it may be tested before the magic number 18 is exceeded. This is achieved by the following:-

100 STACK-BOTTOM !

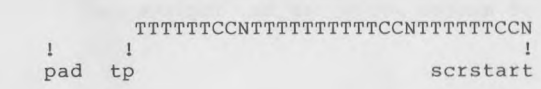
This will tell FORTH to give the error message when the stack pointer drops below 100. Eg.

```
: TEST  1000 0 DO I I . ?STACK LOOP ;
```

If the stack bottom has not been moved then FORTH will crash after the numbers 0-60 have been printed but with the stack set at 100 a 'stack full' message will be given after only 19 numbers have been printed. Since ?STACK needs no parameters it can be added anywhere during the development of an application and then removed when the application is working fully.

Getting Technical

The following diagram shows the layout of screen storage for FORTH.



CC represents the count of characters in the compressed screen. This is a 16 bit number.

N is an 8 bit screen number allowing screens to be numbered 0-255.

T is the compressed text.

Text compresses from SCRSTART with TP (text pointer) decrementing towards low memory, TP will not proceed past the end of PAD, giving an "out of memory" error message.

CONSTANTS, VARIABLES and ARRAYS

The stack is the preferred area for the temporary storage of numbers but for more permanent storage constants and variables are used.

Constants

When a value is frequently used it is a good idea if it is given a name. This increases the readabilty of the program and also reduces the program's size (a call to a constant takes only three bytes but a literal takes five. Remember however that there will be a mimimum of 10 bytes taken up by the constant definition.). Eg.

```
3142 CONSTANT PI
```

This will create the new word PI and assign to it a value of 3142. When PI is later called it will place 3142 onto the stack. Eg.

```
PI 3 * 9426 ok
       -------
```

Variables

A value which changes frequently is called a variable. The FORTH word VARIABLE names a location which can be altered by the programmer. For example it might be necessary to keep a count of the number of faulty components produced during a manufacturing process. Because the fault may happen at a number of times during the processing it is not sensible to leave the count on the stack and so a variable is used in the following manner.

```
VARIABLE ERROR
```

This will create a word called ERROR which will leave the storage address on the stack when ever ERROR is used. At present ERROR has no value. A value is assigned by using the FORTH word ! (store). This will store a 16 bit value at the address on the stack. Thus:-

```
0 ERROR !
```

This will set ERROR to 0.

During processing it will be necessary to find out the current value of ERROR this is achieved using @ (fetch), this replaces the address by its value eg.

```
ERROR @ . 0 ok
          -----
```

Table 5 gives details of address manipulation words. They tend to be short mnemonics and as such take a while to get used to. (Remember that +,-,* and / are mnemonics as well and it would soon become very boring writing 3 5 PLUS 7 DIVIDE each time we wanted to use them just so text would be readable, but talk to a COBOL programmer about that...)

User variables

The user variable is a special variable which is used by the kernal of the FORTH system for holding values which change during the course of use. All of the user variables can be accessed by the programmer but it is not advisable to change them unless you know what you are up to. New user variables cannot be allocated by the programmer but since they behave the same as normal variables there is no need. Here is an example of a user variable being used:-

```
8 DEVICE !
```

The above will make device 8, the disk system, into the storage medium when SLOAD and SSAVE are used.

Arrays

Since VARIABLE leaves the address of a memory location available for storage it is quite a simple extension to turn this into an array by using the word ALLOT . When a new word is created in the dictionary it is always put at the next free area of RAM according to the current position of the dictionary pointer DP . By moving the dictionary pointer along after a variable has been defined an array is created. The dictionary pointer is moved using the word ALLOT which sets aside the number of bytes on the stack eg.

```
VARIABLE TABLE 12 ALLOT
```

This will create a word called TABLE with 14 bytes of storage area (2 bytes are alloted by variable), thus allowing 7x16 bit numbers to be stored in TABLE .

If a lookup table is going to be used it is easier to define this at compile time. Eg.

```
VARIABLE TABLE -2 ALLOT 1 , 10 , 100 , 1000 , 10000 ,
```

-2 ALLOT takes the DP back 2 bytes thus removing the field set up by VARIABLE . The numbers are pushed to the stack and the word , (comma) takes the word and puts it into the next free dictionary space, incrementing the DP by 2 each time. Another word C, is used to store the byte on the stack into the dictionary and move the dictionary pointer along by one byte.

To access an element of TABLE is quite simple:-

```
TABLE 4 + @ . 100 ok
              ------
```

Hints...

One of the most common errors that inexperienced FORTH programmers make is the number of constants and variables which are defined rather than using the stack to hold temporary information. You should always be aware of this and try to utilise the stack.

Getting Technical

In BASIC there is a distinct speed advantage in allocating all numbers used in a program to a variable. This is because BASIC is an interpreter and when it comes across numbers in program it has to convert them to floating point binary. This is a fairly slow process, so it is far better to put all numbers into variables at the start of the program. When a variable name is given BASIC looks it up in its variable table and accesses the floating binary number directly.

When words are being compiled into the dictionary by FORTH numbers encountered in the source code are converted into literals, a literal is 5 bytes long and takes one of the following forms:-

```
     JSR LITERAL  ;call the FORTH word  LITERAL
    .WOR 1000     ;the value of the literal which will be stacked
or
     LDA #250     ;the value is put into the accumulator
     JSR PUSH0A   ;and pushed to the stack
```

When values are less then 256 they are coded in the second form, which is very fast. When values are greater than 255 they are coded in the first form with a call to the word LITERAL which then fetches the value to the stack.

Constants are formed by the following code, as produced by an assembler:-

```
.BYT $86,'co',32          ;name of word
.WOR 1234                 ;link to previous words NFA
 JSR CONSTANT             ;call to constant
 RTS
.WOR 0                    ;the value of the constant
```

A variable is similar in format:-

```
.BYT $86,'co',32          ;name of word
.WOR 1234                 ;link to previous words NFA
 JSR VARIABLE             ;call to variable
 RTS
.WOR 0                    ;the value of the variable, intially 0
```

When CONSTANT executes it leaves the value stored in the PFA, while a VARIABLE leaves the address of the PFA on the stack for later use.

STRUCTURED PROGRAMMING

Structured programming is the means by which a programmer can control the actions of a program using one of the following methods:-

1. Sequential operation where one step follows another.

2. Conditional operation whereby if the condition exists step A is followed, otherwise step B is executed. Both step A or step B continue to step C.

3. Repetitive operation, where step A is continually executed until a condition is met and then it continues to step B.

Conditionals

In BASIC the only condition test is:-

```
10 IF    A=0    THEN B=5: C=7        ELSE B=3: C=5
   IF condition THEN true statements ELSE false statements
```

In FORTH the IF...THEN statement is slightly different:-

condition IF true part ELSE false part THEN continue

ie. The condition to be tested comes before the IF - this takes some getting used to. The condition is true when non-zero and false when equal to zero.

So if the condition is true any statements after the IF and up to ELSE are executed and then the program will jump to the first statement after THEN. If the condition is false then all the code between ELSE and THEN is executed and the program continues after THEN. eg.

```
: TEN  10 = IF ." the number was 10" ELSE ." what was that?" THEN ;

 5 TEN what was that? ok
       -----------------

10 TEN the number was 10 ok
       --------------------
```

The code '10 =' at the start of the definition compares the top stack item with 10 leaving 1 if the numbers are the same and 0 if different. See Table 5 for a full summary of the comparison operators.

The ELSE part of the statement does not have to be present. In this case if the comparison is false then FORTH jumps straight to the code following THEN. Eg.

```
: TEN  10 = IF ." the number was 10" THEN ;

 5 TEN ok
       --

10 TEN the number was 10 ok
      ----------------------
```

Comparisons

There are two types of comparison.

1. Those which test only the top value on the stack eg. 0= (zero equal).

2. Those which compare the top two stack values eg. = (equal).

All the comparison tests destroy their parameters and leave either true (>0) or false (0) on the stack. The comparisons are usually put before IF , UNTIL or WHILE . Often it is unnecessary to use a logical test since the words themselves test for a zero/non-zero result.

Repetitive Structures

There are two types of repetitive structure, those which operate a finite number of times and those which will operate continually or until a certain conditon has been met.

Finite loops

The fastest type of loop is the single incrementing DO ... LOOP .

limit index DO words LOOP continue

Notice that the limit comes before the index. For example:-

: TEST 1000 0 DO R@ . LOOP ;

This will print out the numbers 0 - 999. The limit and index are kept on the return stack with the index being the most accessable value. Note that FORTH drops out as soon as the index=limit and thus does not print out the value 1000.

In many cases it is necessary to work with loops which are decrementing or with an index value altering more than 1. This is achieved by using DO ... +LOOP .

limit index DO words incr +LOOP continue

In this case the index is incremented by the value on the stack before the +LOOP . eg.

: TEST 1000 0 DO R@ . 5 +LOOP ;

This will print out the numbers 0 5 10 15 20 ... 990 995.

A DO ... LOOP can be terminated at any time by using the word LEAVE .

Indefinite loops

There are three types of indefinite loop:-

1. BEGIN xxx condition UNTIL continue

The loop will continually execute xxx UNTIL the condition is true.

2. BEGIN xxx condition WHILE yyy REPEAT continue

xxx is performed and if the condition it leaves on the stack is true it executes yyy and then loops back to xxx again. If xxx leaves a false value then the loop is terminated and execution continues after REPEAT.

3. BEGIN xxx AGAIN

Execute xxx indefinitly. It is only possible to leave the loop with QUIT or ABORT . Or by using the restore key if it is active. See special VIC Functions.

TEXT AND FORMATTED NUMBERS

Character strings are made up from sets of ASCII characters. Such strings are most often used for data I/O.

Formatted numbers are character strings which contain numeric values and any special characters.

String Storage

In general strings are too long to be conveniently handled by the stack and for this reason a temporary storage area called PAD is normally used. The start of PAD is 68 bytes above the current dictionary pointer DP . The amount of free space for the PAD can be found by using the following:-

```
TP @ PAD - . 12323 ok
             --------
```

Remember when using the PAD that it is used to hold lines of text from the editor and that any new dictionary entry will cause the PAD to move. Thus it is not practical to expect strings to remain in the PAD for long periods.

For more permanant storage of strings it is better to ALLOT dictionary space to a VARIABLE as illustrated in the section on tables.

String output

There are two words associated with string output, TYPE and EMIT . TYPE will print an ASCII string and EMIT will print any character value between Ø-255. It is usual that text is followed by a number of unwanted spaces; the word -TRAILING will chop these off, for example:-

```
PAD 2Ø TYPE lots of text        ok
           ----------------------

PAD 2Ø -TRAILING TYPE lots of textok
                      --------------
```

The above example assumes the data is already on the PAD.

The other main word for string output is ." (dot-quote). This prints out the text up to the next quote it encounters, " is not a FORTH word and as such does not have to be separated by spaces:-

```
." This is some text"
```

The distance between quotes is limited to 127 characters.

Moving Data around

To use text effectively it is necessary to move data around in blocks. This is achieved using the FORTH words CMOVE (cee-move) and <CMOVE (reverse-cee-move). The word <CMOVE is non-standard but very useful. The word CMOVE moves text from the end of the string backwards towards the beginning but if there is an overlap in between the two addresses as the text is being moved it will be corrupted eg.

```
abcdefghijklmnopqrstuvwxyz
     !       !
     from    to
```

```
from to 19 CMOVE
```

```
abcdefghijklmfghijklmfghxyz
     !       !
     from    to
```

Notice how the text has overwritten text later in the string. To counter this another word has been defined called <CMOVE this moves data in the opposite direction to CMOVE.

Preparing storage

The main word for setting an area of memory to the same value is called FILL . Two other words BLANKS and ERASE send a space and null (Ø) to FILL . When the PAD is used it is advisable to clear it of previous data:-

```
PAD 64 BLANKS
```

Inputting text and numbers

FORTH is a very flexible language and can quite easily remove the need for menus of options within a program. To allow this the operator has only to remember two simple rules:-

1. Numbers precede the word to be executed.

2. Text follows the word.

The above conventions become second nature to the FORTH programmer, but need to be pointed out to the first time user.

It is however not always convenient to have the data being input prior to executing a word, it is quite often necessary to halt execution, wait for a response and then continue again. To do this a routine called TEXT has been written.

TEXT will allow the user to type any ASCII string up to 64 characters in length, the resulting string is stored at the PAD . The only control keys which have any effect are return and delete, all others are ignored. This makes input totally bomb-proof. Something which leaves a lot to be desired with BASIC....

It can also be very useful to request numbers while a program is running. This is achieved with the word #IN which is a very specific form of TEXT allowing only numbers to be input. A leading minus sign

is possible at the first location position only, otherwise it is ignored. If a decimal point is entered its position will be given by the word DPL .

The word TEXT is aimed towards terminal input. For more general input from other devices the word EXPECT is recommended as this will allow any characters rather than the restricted range of TEXT , also EXPECT can be set to accept any number of characters. Return and null are taken as delimiters and EXPECT will return upon receipt of either.

Number formatting

Numbers are very important in any language and FORTH shows this by providing some very powerful words for formatting numbers prior to being output. It is a simple matter to format numbers so that hyphens, dots and dashes appear in the correct place eg. 3/6/82 as a date.

There are two very simple rules.

1. Numbers are converted left to right.

2. The conversion is performed on double precision values on the stack (32bit). It will usually be necessary to convert 16bit binary into 32bit prior to using the number conversion routines.

16bit unsigned numbers are converted by simply pushing a dummy high 16bit number to the stack Ø. Eg.

```
2000 U. 2000 ok
        -------

2000 Ø D. 2000 ok
          -------
```

If the number is signed use the following:-

```
DUP ABS Ø
```

The 16bit value is duplicated to save the sign, for use by the word SIGN later, the word ABS takes the absolute value of the number, making it positive by setting bit 16, the sign bit, to zero. Finally the zero is pushed to convert the 16bit number into 32bit.

The number conversion words are used in the following order:-

```
<# #S #>
```

<# prepares the PAD for number conversion.
#S converts all significant digits on the stack
this will convert a single digit, if there are no digits, it leaves a zero.
#> terminates conversion, leaving the address of the number and its count.

#S can be used repeatedly to convert different numbers, for example the different parts of the date shown earlier. The word HOLD would be used to add the slashes between the numbers.

The sign of the number can be put either at the start of the number or trailing it, the word SIGN expects a value on the stack which it can test for bit 16 being set.

FORTH AND THE IEEE

The VIC and 64 use a modified version of the IEEE filing system used on the PET. The same file handling syntax has been maintained even though the actual IEEE operations are very different, for example the RS232 channel and the serial IEEE port.

Only six commands are needed to control the IEEE. If you are not familiar with using the bus for file operations then it would be advisable to read the chapters in the manuals for BASIC operation. When you have mastered files in BASIC it is only a small change to use FORTH.

Before a file can be used it is necessary to OPEN it. The OPEN command expects five parameters. These being the logical file number, a device number, a secondary address, an address for a filename and the number of characters the filename contains. If a filename is not required then the count can be set to Ø or 255. Up to ten files can be open at a time.

Audiogenic FORTH does not crash if an attempt is made to open a file which has already been opened, but it is a good idea to always CLOSE files before OPEN is sent, thus clearing any doubt. Eg.

```
15 CLOSE   15 8 15 PAD 5 OPEN
```

When a file has been opened successfully it is necessary to set the direction in which the data is going to travel. This is done by setting the IEEE bus either for input or output. A file cannot be used for input and output simultaneously. It is necessary to set it for I/O in one direction then switch it OFF and reverse the direction. The commands for handling the bus are IN and OUT both have to be given a logical file number, through which communications are to take place. The command OFF restores the default I/O, ie. the keyboard and the screen. Before communicating with a different logical file it is necessary to switch the bus OFF . Eg.

```
1 IN  ØKEY  OFF  4 OUT  EMIT  OFF
```

Always remember to switch the bus OFF before returning control to the user otherwise the only key that will usually work is the RESTORE key!

The two special commands SLOAD and SSAVE , which save and load the source code, will work with any device. The word DEVICE is set to 1, for cassette deck operation, when FORTH is powered on. To change the device to say disk, simply type the following:-

```
8 DEVICE !
```

The command STATUS will push the current status of the system to the stack. This is the same as given for BASIC.

Hint...

When IEEE files are going to be used it is best to leave the file I/O until last, simulating the IEEE operation by taking input from the keyboard and output to the screen. Only when everything is de-bugged should the actual files be worked with because usually they have their own set of problems which can cloud the programming if tackled at the same time as the main logic of the program.

Giving a file name to the open command can cause problems. If the user is going to enter the filename there are two courses of action open to the programmer.

1. Use the word TEXT . This puts a small arrow onto the screen and lets the user enter any ASCII information. The information being entered is put at PAD with the number of characters being entered as the first byte, the following can then be used to open the file:-

```
1 1 1 TEXT PAD COUNT OPEN
```

The word COUNT is very useful as it expects an address on the stack. When it executes it finds the byte stored at the address and also adds one to the address, thus leaving the count and the real address of the text.

2. This method can only be used when the user types the word at the keyboard followed by the filename:-

```
: READ 1 1 Ø 32 WORD HERE COUNT OPEN ;
```

when READ is used the following is typed:-

```
READ DATA
```

This will open a file called data on the cassette ready for reading. The command 32 WORD is the way in which FORTH moves a word from the TIB to the current position of the dictionary pointer (you are in fact accessing one of the compiler words). The word HERE pushes the current dictionary pointer to the stack. If instead of 32 WORD you had 34 WORD the following would be required before FORTH accepted the input:-

```
READ DATA"
```

ie. it would then look for a double quote (")

3. Very often it is necessary to access a filename which the user is not expected to enter. This is a little bit more difficult, here is one method:-

```
: FILENAME ." test-file" ;
: READ 1 1 1 ' FILENAME COUNT OPEN ;
```

The word ' searches the dictionary for the word FILENAME and returns its PFA which by luck holds the count for the number of characters in the string.

WRITING A FORTH PROGRAM

FORTH is an interactive language which means that you will spend most of your time at the keyboard working on new ideas, unlike more conventional compilers where different programs have to be used to generate intermediate stages of code, for example, editors, linkers and loaders. This interactive method of programming is far superior to the more conventional approach, and this is one of the main reasons why BASIC is so good as a first time language. While Pascal has a lot going for it in the way of structure and data types it is hard to believe that the computer revolution would have occurred if that were the only language available, due to the time it takes to compile/debug large programs.

To write a FORTH program all you have to do is sit down and think about it. The main question you have to ask yourself is 'What are the operations involved?'. These operations become FORTH words, each of which can be debugged separately. This is very important. Defining a mass of words and then trying to debug them all together is asking for trouble.

It may help you if I describe how I approach putting together a FORTH program. The program I will describe is the longest given in this manual, it is an arcade type game called MineSweeper. First of all I obviously need the idea for the program and I usually think about it for a couple of days before sitting down and attempting to put anything together. This gives me a chance to think out the various sections of the program which are going to be necessary. Because of the way in which FORTH works it is not necessary to put all the refinements in at once, for example I added sound and the words ATTACK and MAX? , to the game at a very late stage.

I wrote Screen 2 first as it provides a framework on which to build. Screen 3 came next and gave the most trouble in the whole program. It took three system crashes due to the stack overflowing before I realised that DROP is necessary in the word CHANGE . The usual practice of using the word once and then testing the stack to see that nothing was left does not work in this case as the program only gets to the IF DIRECTION ! DROP THEN when a cursor key has been pressed and the routine is so fast that to run it less than a thousand times just does not give you chance to hit a key!

The basis for the game now exists, the ship will move around the screen under the control of the cursor keys. The next stage is to get some action when the ship moves over a bomb or enemy ship. DAMAGED give the ship three chances of bumping into the bombs but rather than give just the normal three ships an actual damage report is printed at the top of the screen depending on the value of SHIELDS . Remember that MAX? was added much later. HITALIEN adds a value to the score which is displayed and moves the alien to another position on the screen. The only other routine which now has to be produced is the one which moves the alien to another part of the screen if you can not ram him in time, this is included in ONETIME . The game is now finished, the rest is just icing.

From the above it should be obvious that I wrote quite a few versions of the game before arriving at the final one. I added the sound and the ability for your ship to run over the mines without damage at some stages as new versions and MAX? was added as an after-thought to demonstrate text handling.

Screen 6 was the most often changed of the five screens but usually only by additions to it. It started out being just a call to SETUP and then a DO ... LOOP containing MOVE and CHANGE and as other words were added so Screen 6 was modified to take account of them.

The main principal behind developing any program is to know exactly what is on the stack at any time, especially values which are being passed from word to word. The reason the system crashed while programming Screen 2 was simply that I did not know exactly what the program was doing at all stages. To have programmed it without this error I should have used the stack to its best advantage. Instead of waiting for a key to be pressed the word ØKEY should have been removed and a value for KEYS taken from the stack. Eg.

```
: CHANGE  KEY ! Ø     etc.
```

I could then have put different key values onto the stack and tested the word thoroughly. When it was working without exceptions the word ØKEY could be put into position.

There is no real excuse for crashing FORTH and with most words it is robust enough to stand up to errors; it is just the odd time when a slight slip or even lazyness can cause the system to malfunction.

I will now try to describe what I call stack diagrams. They are more common sense and an aid to the memory than anything and as such are rather difficult to define. A very simple stack diagram is shown with each word definition in the tables eg:-

```
ROT  ( n1 n2 n3 --- n2 n3 n1 )
```

This shows that ROT expects 3 values on the stack before it is called with n3 being the most accessible. When the word has finished executing, n1 has been moved to the most accessible position rather than the least accessible.

This needs extending if it is going to be of use. Take for example the word DOT which forms part of the VIC hi-res screen described elsewhere.

```
: DOT     16 /MOD  336 *  4Ø96 + +
     SWAP  8 /MOD   16 *  ROT  +
     DUP   C@ ROT SET OR  SWAP C! ;
```

The overall stack diagram for DOT is:-

```
DOT  ( x y --- )
```

i.e. it expects an x,y coordinate on the stack when it starts, so let's start there:-

```
x y 16          (values on stack)
    16          (operations)
```

```
x rem(y/16) int(y/16) 336
                /MOD 336

x rem(y/16) int(y/16)*336 4096
                       * 4096

x rem(y/16)+int(y/16)*336+4096
                         + +

rem(y/16)+int(y/16)*336+4096 x  8
                          SWAP  8

yval rem(x/8) int(y/8) 16
                /MOD 16

yval rem(x/8) int(y/8)*16
                       *

rem(x/8) int(y/8)*16+yval
                    ROT +

rem(x/8) add add
           DUP

rem(x/8) add val
            c@

add val rem(x/8)
           ROT

add val bit
       SET

add valbit
       OR

valbit add
      SWAP

C!
```

In reality the stack is not rewritten for every operation, values are simply crossed out and rewritten further along the line. Only when the stack diagram gets confusing is it rewritten.

When using the diagrams it is necessary to produce a new one each time a logical decision is made. If the return stack is being used I usually keep another diagram for its operations.

In general the stack diagrams are unnecessary. I did not use one while I developed MineSweeper, and it is only when many operations are being grouped together that they become very useful.

USER DEFINED DEFINING WORDS

The user defined defining words are one of FORTH's most powerful features, they are also one of the most difficult to understand.

Three defining words are in common use, they are : (colon) CONSTANT and VARIABLE . They are all used to compile new words into the dictionary which have the same general effect but are unique in their action.

The colon definition will create a named entry in the dictionary which will compile the following word's CFAs into the dictionary as a series of subroutine calls. When a colon definition is terminated a return from subroutine (RTS) is compiled into the dictionary.

CONSTANT and VARIABLE are easier to understand because they are not as general as colon. Both the CONSTANT and the VARIABLE occupy the same amount of space in the dictionary and both keep a value in the PFA , so where is the difference? It is obviously in the word which is compiled into the dictionary to access the PFA . CONSTANT compiles a word which fetches the value at the PFA and pushes it to the stack, while VARIABLE compiles a word which only pushes the address of the PFA to the stack.

```
: CONSTANT  <BUILDS    , DOES> @ ;
: VARIABLE  <BUILDS Ø , DOES>    ;
```

The above shows the definition of both CONSTANT and VARIABLE in terms of the words <BUILDS and DOES> the high level difining words. In reality special high speed words have been designed for both these common data types but the concept of <BUILDS and DOES> is displayed well.

When CONSTANT is used in future it is called with the following syntax:-

```
12 CONSTANT MONTHS
```

The word <BUILDS will create the dictionary entry MONTHS . The value on the top of the stack is then put into the PFA of MONTHS . Now comes the difficult bit. DOES> modifies the CFA of MONTHS to point to the code following DOES> but it is modified in such a way that the PFA of MONTHS is put onto the stack in the process! Thus when MONTHS later executes, the PFA will be pushed to the stack, and the word @ (fetch) then gets the values stored at the PFA .

The word VARIABLE performs in much the same manner:-

```
VARIABLE DATE
```

<BUILDS compiles a dummy value for the variable into the dictionary, the most obvious being a zero but it could be anything. DOES> has no words following it so it simply leaves DATE 's PFA behind on the stack.

Getting Technical

Before describing the code involved with DOES> it is worth first of all reading the section on how the compiler works, paying special attention to the PFA .

<BUILDS is actually the following definition:-

```
: <BUILDS  Ø CONSTANT ; IMMEDIATE
```

I.e. It creates a CONSTANT in the dictionary! This is just for convenience as the code will be modified by DOES> . Taking the definition of VARIABLE as shown above as an example, the code so far looks like:-

```
.byt $86,'co',12
.wor $2121
JSR DOCON              (1)
RTS
.wor Ø                 (2)
```

Any words between <BUILDS and DOES> are then compiled into the dictionary following the above code. With VARIABLE this will be zero as a 16bit value (ie Ø is pushed to the stack and , (comma) compiles the number into the next available dictionary space, incrementing the dictionary pointer by two. The code now looks like:-

```
.byt $86,'va',12
.wor $2121
JSR DOCON              (1)
RTS
.wor Ø                 (2)
.wor Ø                 (3)
```

DOES> changes the address following the JSR at (1) to point to a system word called DODOES (which is not accessible as a dictionary entry). The zero which was stored at (2) is overwritten with the address of the code which followes DOES> in the main definition of the word. In this case it simply points to the RTS at the end of the word.

```
.byt $86,'va',12
.wor $2121
JSR DODOES             (1)
RTS
.wor addrl             (2)
.wor Ø                 (3)
```

FORTH does not have matrix manipulation words built in, so here are a few matrix data types to demonstrate <BUILDS and DOES> .

The simplest matrix is one which creates a vector (i.e. a 1 dimensional array) of bytes. It is used in the form:-

```
365 CVECTOR DAYS       (4)
```

This will create the word DAYS as a vector of 365 bytes.
To access a byte the following is used:-

```
30 DAYS C@ .          (5)
```

The above would fetch the 30th byte and display it on the screen. The definition for CVECTOR is:-

```
: CVECTOR  <BUILDS  1+ ALLOT  DOES> + ;
```

When the vector is created at (4), 366 bytes of dictionary space are alloted to the word definition DAYS and the runtime code is pointed to the value after DOES> . It is necessary to allocate one more byte using 1+ because otherwise the matrix will only work in the range 0 to 364 which occupies 365 bytes. When DAYS is later used (5) the value on the stack is added to the PFA which is already on the stack and the modified is left for the user to work with.

Here is a vector for 16bit values. It is created and used in the same manner as above:-

```
12 VECTOR MONTHS      (6)
30 MONTHS @ .         (7)

: VECTOR  <BUILDS 1+ 2* ALLOT  DOES>  SWAP 2* + ;
```

Because ALLOT moves the dictionary pointer along by the number of bytes on the stack it is necessary to double the number so that 16bit cells can be left, once again the value on the stack is incremented by one to allow access to the twelfth position of the vector. When the word executes (7) the PFA being left by DOES> is swapped with the value on the top of the stack, 30, which is multiplied by two to index to the correct cell.

Here is a more complex example, it is a two dimensional byte matrix and is used in the form:-

```
10 20 CMATRIX TABLE    (8)
5 3 TABLE C@ .         (9)

: CMATRIX  <BUILDS   1+  DUP , SWAP 1+  * ALLOT
             DOES>  DUP @ ROT * + + 2+ ;
```

When the word TABLE is created (8), 231 bytes of dictionary space have to be alloted simply by multiplying the two maximum values together, one added to each value. However it is also necessary for TABLE to know the size of one dimension (from this the other one can be inferred). The dimension is thus stored in the first 16bits of the array. When the word later executes (9), the PFA is pushed to the stack as normal. At the PFA resides the dimension (in this case 20), this is brought to the stack and multiplied by the high index, 3 in this example and then the PFA and the low index, 5, are added to this giving a unique position in the array. The 2+ at the end of the DOES> moves the address along by 2 bytes to skip over the PFA . Modifying location 0 0 and 1 0 would destroy the dimension stored at the PFA if this was not done.

FORTH MEMORY ORGANISATION

VIC FORTH - cartridge including 3k of RAM

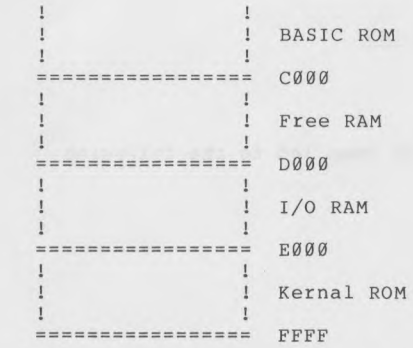
When VIC-FORTH has been purchased with 3k of RAM included as standard two different memory maps are possible. The first shows the version with only the 3k of RAM in the cartridge. The second memory map shows how VIC-FORTH is reconfigured when the cartridge is used in a motherboard with extra memory of 8K or 16K.

RAM used to 0400 is the same with both systems
0400
FORTH editing screen
0803
User variables
082C
free RAM
1000
video RAM
13FF
VIC-FORTH SYSTEM WITH 3k OF INTERNAL
MEMORY & 8 or 16k OF EXTERNAL MEMORY
<---dictionary pointer (increments)
PAD 64 bytes above the dp
free RAM
<---text pointer (decrements)
Top of memory

VIC FORTH - cartridge without RAM

The version of the VIC FORTH cartridge without RAM must be used in a motherboard in conjunction with 8K or 16K expansion. The memory map looks like this...

FORTH - Commodore 64 version



The stack pointer decrements by two bytes when data is pushed to either the parameter stack or to the return stack and increment by two bytes when data is removed.

When an application is running, the area used by the FORTH editing screen can be used to hold temporary data such as tables or strings; the current contents of the screen will be destroyed but since this will normally be screen 0, it hardly matters.

The dictionary pointer and the text pointer move through RAM towards each other. The text pointer will not drop beyond the address of the PAD , the dictionary pointer does not follow this restriction and because of the way in which the text is saved in RAM an application will usually load correctly even though the source code becomes corrupted in the process. Because of this and the fact that it is not very difficult to crash FORTH, saving the source code at regular intervals is highly recommended.

HOW THE COMPILER WORKS

Take the following word definition:-

```
: TEST DUP DROP ;
```

This is a very simple definition and would be compiled to the following machine code:-

Name	Code	Note
NFA	$84 te $a	(1)
LFA	$2021	(2)
CFA	jsr dup	(3)
	jsr drop	
	rts	(4)

Fig 1.

(1) Colon (:) turns the compiler on and searches the dictionary for the following word. If the word is found then the error message 'test isn't unique!' is printed. Compilation continues and any references to 'test' will access this version rather than earlier versions of 'test'.

The length of the word being defined is stored in the first byte of the definition. The high bit of the count is set to help with converting the PFA (Parameter Field Address) to its NFA (Name Field Address).

Some words are defined so that they are executed rather than compiled when FORTH is compiling a new word into the dictionary; such words are termed IMMEDIATE . For a word to be IMMEDIATE bit 6 is set. Words such as IF , ELSE and THEN are IMMEDIATE because they have to perform calculations on where they are to branch to after the logical tests. If they were not IMMEDIATE they would simply compile to a subroutine call (see note 3).

If a word fails to compile properly due to a programming error then bit 5 is set. This bit is called the smudge bit. The smudge bit is included in the length of the word, thus a five letter word would become a 37 letter word if compilation failed.

The number of letters in the word is stored in bits 0-4 giving a maximum of 31 letters.

```
7 6 5 4 3 2 1 0     <-- bit number
1 0 0 0 0 1 0 0     <-- example value
! ! ! < count >
! ! !
! ! set if compilation failed
! set if immediate
always set
```

Fig 2.

If the word being defined has a name which is less than 4 characters long then only those characters are stored. Eg. A , BE , CEE

When the word has a length greater than 3 characters then the exclusive-OR of all the letters is taken and stored after the first two characters of the name. The exclusive-OR has a very strange feature. So long as the first two letters remain the same the rest of the word can be in any order and it will still be found in the dictionary. Eg:-

IMMEDIATE IMETAIDEM IMATEEDIA

This cures one of the big problems of FORTH systems which do not maintain the full name. For instance it is quite common to give words names like:-

TEST1 TEST2 TEST3

In many FORTH systems this would all be taken as being the same word as only the length and the first three letters are maintained.

(2) Each word points back to the start of the previous word using a 16 bit pointer. This linked list structure makes FORTH a threaded language.

(3) In standard FORTH systems there are two interpreters. Firstly, an outer interpreter which processes words typed in at the terminal and loaded from screens. Secondly, an inner interpreter which processes a list of addresses pointing eventually to machine code. The inner address interpreter governs the speed at which FORTH can run.

By compiling right down to machine code the inner interpreter has been scrapped allowing the language to run faster. The only drawback to this is that each word compiled into the dictionary takes an extra byte (JSR), speed has to be paid for by slightly larger definitions. The standard inner interpreter takes approximatly 130 machine cycles, the JSR-RTS structure takes 12, making JSR-RTS roughly 10 times faster.

(4) A definition is terminated by a simple return from subroutine (RTS) rather than a call to the inner interpreter as with standard FORTH. ; also switches the compiler off and returns control to the interpreter. The header is also altered (bit 5 of Fig 1 is cleared) making the word available in the dictionary.

In Fig 1. various fields have been given names.

NFA - Name Field Address. The address of the definitions name. Always points to the count.

LFA - Link Field Address. The address of the link field which points back to the last word in the dictionary.

CFA - Code Field Address. The address of the first byte of code. It will always point to a 6502 instruction, usually JSR.

PFA - Parameter Field Address. In Fig 1. a field was missed out - the PFA . The PFA is the address of the first byte of memory available for storing parameters in a word. However the PFA is always five bytes after the start of the code in a word and thus has little meaning in most word definitions. It only really has meaning when used with variables and constants. eg.

```
Name        Code

NFA         $83,'TRY'
LFA         $2021
CFA         jsr variable
            rts
PFA         2000
```

When the word TRY is typed, the call to the subroutine variable causes the PFA to be put onto the stack. The value at this address can then be found by using the following:-

```
TRY @ . 2000 ok
        -------
```

REFERENCE TABLES

The following tables contain information on every word which has been defined in the FORTH system. The tables are intended mainly for reference, but the first time user may find it worthwhile to read through the following:-

Table 3 gives all the information you need for working the editor.
Table 8 details all of the compiler commands.
Table 10 shows how to save and load cassette files.
Table 11 shows the error messages you will encounter.

List of Tables

Page	Table	Description
4	Table 1	- Stack manipulation
6	Table 2	- Mathematics
9	Table 3	- Screen manipulation
11	Table 4	- Editor commands
13	Table 5	- Address operations
14	Table 6	- Logical operators
15	Table 7	- Dictionary control
18	Table 8	- Compiler directives
21	Table 9	- Input/Output
23	Table 10	- IEEE
25	Table 11	- Error messages
27	Table 12	- Numeric Formatting
28	Table 13	- Defining words
29	Table 14	- Structure words
31	Table 15	- Special VIC/64 commands

The following letters are used to designate stack values:-

n	16bit signed number	(-32768 to 32767)
u	16bit unsigned number	(0 to 65535)
b	8bit number	(0-255)
c	ASCII character	(0-255)
d	32bit signed number	(-2,147,483,648 to 2,147,483,647)
ud	32bit unsigned double	(very big!)
t/f	true or false	(0 false 1 or greater true)
addr	Memory address	(0 to 65535)

Words in the dictionary

Stack operations (Table 1)

OVER DROP SWAP DUP ROT ?DUP
SP@ SP! RP! >R R> R@ I J K STACK-BOTTOM

Maths and numbers (Table 2)

DIGIT + D+ MINUS DMINUS MIN MAX AND OR XOR U* U/ 1+ 2+ 1-
2- - HEX DECIMAL LITERAL NUMBER S-->D +- D+- ABS DABS * 2*
2/ MOD /MOD M/MOD / M* */ */MOD M/MOD Ø 1 2 3 -1 DPL
BASE

Screen storage (Table 3)

TP SCR BLK EXP COM EXPAND SSAVE SLOAD DEVICE ?SCREENS DELETE
COPY RESET CLRØ --> (LOAD SCREEN SCRSTART C/L CONTROL ←

Editor (Table 4)

All the control functions of the FORTH full screen editor.

Memory Operations (Table 5)

FILL ERASE BLANKS CMOVE <CMOVE @ C@ ! C! TOGGLE +! , C,
COUNT

Logical Operators (Table 6)

Ø= Ø< - = U< < >

Dictionary Operators (Table 7)

FENCE SET-FORTH -FIND HERE ALLOT , C, LATEST TRAVERSE LFA CFA
NFA PFA FORGET ' +ORIGIN PAD VOCABULARY DEFINITIONS DP
CURRENT CONTEXT

Compiler directives (Table 8)

>IN EXECUTE ENCLOSE ERROR ?ERROR ?STACK ?COMP ?EXEC ?PAIRS
?CSP ?LOADING COMPILE [COMPILE] [ ] WORD CREATE !CSP : ; ;S
QUIT ABORT INTERPERET IMMEDIATE STATE SMUDGE COLD LITERAL

I/O (Table 9)

KEY ØKEY EMIT CR TYPE -TRAILING (.") ." EXPECT QUERY SPACE SPACES D.R D. .R U. . ? CLR ?TERMINAL TEXT #IN P HOME BL TIB

IEEE (Table 1Ø)

OPEN CLOSE IN OUT OFF STATUS

Error Messages (Table 11)

A list of FORTH's error messages.

Numeric formatting (Table 12)

HOLD <# #> SIGN # #S

Defining words (Table 13)

CONSTANT VARIABLE <BUILDS DOES>

Structure (Table 14)

BRANCH ØBRANCH (LOOP) (+LOOP) (DO) LEAVE DO LOOP +LOOP BACK BEGIN UNTIL AGAIN IF ELSE THEN REPEAT WHILE DOBRAN I J K R@

Special VIC/64 functions (Table 15)

SOUND VOLUME NMI COLOUR SET RESTORE

Table 1 - Stack Manipulation

Word	Stack	Effect
SWAP	(n1 n2 --- n2 n1)	Swap the top two values.
DUP	(n1 --- n1 n1)	Duplicate the top value.
?DUP	(n1 --- n1 n1) (0 --- 0)	Duplicate the top value only if non-zero.
DROP	(n1 ---)	Drop the top value.
OVER	(n1 n2 --- n1 n2 n1)	Push a copy of the second stack value to the top.
ROT	(n1 n2 n3 --- n2 n3 n1)	Rotate the third stack value to the top of the stack.
>R	(n1 ---)	Transfer value to the return stack.
R>	(--- n1)	Transfer value from the return stack.
R@	(--- n1)	Copy the top of the return stack.
I	(--- n1)	Copy the top of the return stack. Used to fetch the loop index. Same as R@ . DO J . LOOP
J	(--- n1)	Copy the third item of the return stack to the parameter stack. Used to fetch a second loops index. DO ... DO J . LOOP LOOP
K	(--- n1)	Copy the fifth item of the return stack to the parameter stack. Used to fetch a third loops index. DO ... DO ... DO K . LOOP LOOP LOOP

SP@	(--- n1)	Leave the current value of the stack pointer.
SP!	(---)	Reset the stack pointer to its initial value.
RP!	(---)	Reset the return stack to its initial value.
STACK-BOTTOM	(--- addr)	A user variable which sets the bottom of the parameter stack. When the system powers on STACK-BOTTOM is set to 18. This is the absolute maximum depth of the stack. If values exceed this depth FORTH stands a very good chance of crashing. When de-bugging programs set the STACK-BOTTOM to around 80 to help trap errors quickly. 80 STACK-BOTTOM !

Numeric Stack Output

.	(n ---)	Print n as a signed value, followed by a space
?	(addr ---)	Print the contents of the address as a signed value followed by blank.
D.	(d ---)	Print the 32 bit value as a signed value followed by a space.
U.	(n ---)	Print the value as an unsigned value followed by a space.
.R	(n1 n2 ---)	Print the signed value n1 right-aligned in a field of n2 characters.
D.R	(d n ---)	Print the signed 32 bit value right-aligned in a field of n characters.

Table 2 - Maths and Logical Operators

Word	Stack	Effect
1. Single Precision Operators		
+	(n1 n2 --- sum)	Add two values
+-	(n1 n2 --- n3)	Assign the sign of n2 to n1 and leave as n3.
-	(n1 n2 --- diff)	Subtract two values
1+	(n --- n+1)	Add 1
2+	(n --- n+2)	Add 2
1-	(n --- n-1)	Subtract 1
2-	(n --- n-2)	Subtract 2
*	(n1 n2 --- product)	Multiply
2*	(n --- n*2)	Multiply n by 2 using byte rotate
2/	(n --- n/2)	Divide n by 2 using byte rotate
/	(n1 n2 --- quotient)	Divide n1/n2
MOD	(n1 n2 --- rem)	Leave the remainder of n1/n2
/MOD	(n1 n2 --- rem quot)	Divide leaving both remainder and quotient
*/	(n1 n2 n3 --- quot)	Multiply n1 by n2 and divide by n3. (n1*n2)/n3 This is called star-slash.
*/MOD	(n1 n2 n3 --- rem quot)	As */ but leaves remainder as well
ABS	(n --- u)	Take absolute value.
MINUS	(n1 --- n2)	Change sign
MIN	(n1 n2 --- n3)	Leave the smaller of n1 and n2
MAX	(n1 n2 --- n3)	Leave the larger of n1 and n2
AND	(n1 n2 --- n3)	Leave the logical AND of n1,n2
OR	(n1 n2 --- n3)	Leave the logical OR of n1,n2
XOR	(n1 n2 --- n3)	Leave the Exclusive OR of n1,n2

Word	Stack	Effect
2. Double Precision Operators		
D+	(dl d2 --- dsum)	Add two 32 bit values
D+-	(dl n --- d2)	Assign dl the sign of n leaving d2
DMINUS	(dl --- d2)	Change the sign of dl
DABS	(d --- ud)	Leave the absolute value of d
3. Mixed-Precision Operators		
S->D	(n --- d)	Convert a 16 bit value to a double number.
M/	(ud n --- nrem nquot)	Divide unsigned double 32 bit by signed 16 bit leaving signed results.
M*	(nl n2 --- d)	Multiply two 16 bit numbers giving a 32 bit result
M/MOD	(ud n --- urem uquot)	Divide the unsigned 32 bit by the unsigned 16 bit leaving unsigned results
U/	(udl ul --- urem uquot)	Divide the unsigned 32 bit value by an unsigned 16 bit value giving unsigned remainder and quotient.
U*	(ul u2 --- ud)	Multiply two unsigned 16 bit numbers giving an unsigned double

Number conversion

DIGIT (c n1 --- n2 t) conversion ok
(c n1 --- f) conversion failed.
Converts the ASCII character c into binary according to the base n1. If conversion is okay the binary value and a true flag are left on the stack. If conversion fails a zero is left.

NUMBER (addr --- d)
Converts the number at the address into a double number according to the current value in BASE . The first byte at the address is the length of the number to be converted. If the number conversion fails due to a non-numeric character being present an error message is given eg:

12.3er say what?

DPL (--- addr)
A user variable which holds the position of the decimal point in the most recently converted number. If the number does not contain a decimal point then DPL is set to -1.

BASE (--- addr)
A user variable containing the current numeric input/output conversion base. Note that all numbers are held in the computer in binary.

HEX (---)
Sets BASE to hexadecimal I/O, base 16.

DECIMAL (---)
Sets BASE to normal decimal I/O, base 10.

0 (--- 0)
Push 0 to the stack

1 (--- 1)
Push 1 to the stack

2 (--- 2)
Push 2 to the stack

3 (--- 3)
Push 3 to the stack

-1 (--- -1)
Push -1 to the stack

Table 3 - Screen Storage

← (n ---)
Edit Screen n. Compress screen Ø and expand screen n. If the screen has not been edited before, an erased screen is set up ready for editing.

BLK (--- addr)
A user variable which points to the screen being edited or interpreted. If zero the text is taken from the keyboard.

SCR (--- addr)
Only used on the VIC-FORTH system to show the last screen edited, pressing F2 will automatically bring the screen back.

COM (---)
Compress the current screen to RAM storage, the screen being compressed is numbered according to the value of BLK . An 'out of memory' message is displayed if the text pointer TP is less than the end of the temporary storage area PAD . Try and avoid having screens with the same number, it can get very confusing.

EXP (---)
Search through the RAM storage for the screen number in BLK . If the screen is found then it is expanded and the compressed screen is deleted from storage. When the screen is later compressed it is placed at the end of the RAM storage area according to the value of TP . If the screen is not found a blank screen is set up ready for editing. Eg. 1 BLK ! EXP this will expand screen 1 from storage.

EXPAND (n ---)
Compress the screen to RAM with the value stored in BLK then expand screen n. Beware of using this command while interpreting and always reselect the screen which originally contained the command. The command is made up of the following FORTH words COM BLK ! EXP).

SCRSTART (--- n)
A constant which points to the start of storage RAM. TP is initialised to this value.

RESET (---)
Resets the value of TP to SCRSTART having the effect of clearing the source storage RAM. The dictionary is not affected.

TP (--- addr)
A user variable which points to the next available position for text compression.

PAD (--- addr)
Leave the address of PAD , this area is 64 bytes above the current dictionary pointer DP and can be used for temporary storage of data. Also used by the line editing commands.

?SCREENS (---)
Prints the screen numbers currently held in RAM.

COPY (n1 n2 ---)
Copy screen n1 to n2. If n1 does not exist then a blank screen is copied. n2 can be a duplicate screen number but it will not be accessable until the other screens with the same number have been accessed.

DELETE (n ---)
Delete screen n from storage. When deleted the screen overwrites the contents of screen Ø.

CLRØ (---)
After a lot of use Screen Ø will get very cluttered. The command CLRØ can be entered to erase the contents of the editing screen.

C/L (--- n)
A constant which leaves the number of characters per line on the screen; it will leave 64.

((---)
Parenthesis is a word which allows comments to be embedded into source code. When the compiler finds a parenthesis is skips text until it comes to another closing parenthesis, where it resumes operation eg.
(THIS IS A COMMENT)

CONTROL (--- addr)
This User variable contains the address of code which will execute when the function key F1 is pressed. See details at end of this table.

LOAD (n ---)
Load screen n by invoking the interpreter. LOAD is used to compile applications stored in screens. LOAD can be used within a screen definition. When the screen has been loaded FORTH returns to the screen from which it was called.

--> (---)
Stop compiling/interpreting the current screen and proceed to the next one. This is much more elegant than LOAD within screen definitions.

SLOAD xxx (---)
This command is followed by a filename eg:-
SLOAD programl
This will load the source code called 'programl' from cassette storage to the current position of TP , thus allowing different sections of code to be appended together.

SSAVE xxx (---)
This saves all the screens stored in RAM to cassette storage, or to another device specified by the word DEVICE using the same syntax as sload. Eg.
SSAVE sourcecode (cassette storage)
SSAVE @0:sourcecode,s,w (disk storage)

DEVICE (--- addr)
This is a User variable which contains the current storage device. When the sytem is switched on device is set to 1 for cassette operation. When device is changed to 8 the source code will be loaded/saved to disk.

For more detailed information on SLOAD and SSAVE see Table 1Ø - IEEE

Table 4 - Single key Editor commands

These commands are not available as FORTH words.

cursor left --- move the cursor 1 position to the left, scrolling the screen if necessary. If the cursor is at column zero wrap around to the previous line.

cursor right --- move the cursor 1 position to the right, scrolling the screen if necessary. If the cursor is at position 64 wrap around to the next line. Where possible the screen starts to scroll 5 characters before the right edge of the screen so as to give some look-ahead.

cursor down --- move the cursor down to the next line. It will not move past line 15.

cursor up --- move the cursor up one line. It will not move beyond line Ø.

insert --- move the text at the cursor one space to the right inserting a space at the current position. If position 64 is filled then insert does not work.

delete --- move the text at the cursor one space to the left. It will not move beyond column zero.

home --- move the cursor to line Ø column Ø.

clr --- same as home.

return --- If return is pressed while on screen zero the interpreter is invoked. On any other screen return moves simply to the next line.

shift return --- Behaves like return except everything from the cursor position to the end of the line is erased.

The following commands are only available on VIC-FORTH and are accessed by pressing the function key.

F2 --- Hold the current line in the PAD.

F3 --- Spread the text from the current line. Line 15 is lost. A blank line is left at the current line.

F4 --- Spread the text and then insert the contents of the PAD at the current line.

F5 --- Delete the current line, holding it in PAD and move the rest of the text up. Line 15 is left blank.

F6 --- Replace the current line with the contents of PAD.

F7 --- Exit from the current edit screen back to screen zero, compressing the screen to storage and expanding screen zero from storage. The cursor position remains the same.

F8 --- Re-edit the last screen accessed.

F1 --- This is a special VIC-FORTH function key to which you can assign your own functions. If you assign nothing to F1 it still has a very powerful function ... it prints out the current contents of the temporary storage area PAD . The screen editor is simple to use but sometimes it is useful to see the entire line being edited. Simply press F2 then F1 with the cursor resting on the line you wish to see. It will be displayed at the bottom of the screen.

To assign your own functions you first of all need to define the word as part of the dictionary eg.

```
: FREE   Ø 19 P  TP @  PAD 64 + - . ;
```

FREE will show the amount of totally free space available for new definitions. The value is printed at the bottom of the screen.

To allocate the word to the function key is very simple:-

```
' FREE CFA CONTROL !
```

' (tick) finds the word in the dictionary. CFA converts the PFA (parameter field address) which has been left on the stack by ' into the CFA (code field address). The CFA is then stored in a user variable called CONTROL .

Here is another example:-

```
: STACK Ø 19 P   SP@ . ;
```

This will print out the current value of the stack pointer at the bottom of the screen. This is a very useful debugging tool. You should always know how many parameters are being left/used by a word, otherwise there is a good chance the system may crash.

Table 5 - Memory Operations

@ (addr --- n)
Replace the address by its contents. (fetch)

! (n addr ---)
Stores the 16 bit number at the address on top of the stack. (store)

C@ (addr --- b)
Replace the address by the 8 bit byte at the address. (cee-fetch).

C! (b addr ---)
Store the 8 bit byte at the address. (cee-store).

+! (n addr ---)
Increments the contents at the address by the value n.

, (n ---)
Store n at the current dictionary pointer and then move DP along by 2. (comma).

C, (b ---)
Store b at the current dictionary pointer and increment DP by one. (cee-comma).

FILL (addr n b ---)
Fills n bytes with value b starting at the given address.

ERASE (addr n ---)
Sets n bytes to value Ø starting at the given address.

BLANKS (addr n b ---)
Sets n bytes to spaces starting at the given address.

CMOVE (addl add2 n ---)
Moves n bytes from addressl to address2 starting at addressl.

<CMOVE (addl add2 n ---)
Moves n bytes from addressl to address2 starting from the addressl+n

TOGGLE (addr b ---)
The byte at addr is XORed using byte b as a mask.

COUNT (addr --- addr+1 u)
Mainly used for text manipulation where the first byte, as given by addr is the length of the text.

Table 6 - Logical Tests

Word	Stack	Effect
=	(n1 n2 --- t/f)	Leaves true if n1 = n2
-	(n1 n2 --- diff)	Leaves true if n1 <> n2
<	(n1 n2 --- t/f)	Leaves true if n1 < n2
>	(n1 n2 --- t/f)	Leaves true if n1 > n2
U<	(u1 u2 --- t/f)	Leaves true if u1 < u2 : U< Ø SWAP U/ Ø= SWAP DROP ;
Ø=	(n --- t/f)	Leaves true if n=Ø
Ø<	(n --- t/f)	Leaves true if n is negative

True is any non-zero value, but in the cases above where t/f has been left on the stack true will be 1.
False is Ø.

Table 7 - Dictionary Management

DP	(--- addr)	Leaves the address of the user variable containing the current value of the dictionary pointer.
HERE	(--- dp)	Push the current value stored in the dictionary pointer.
PAD	(--- addr)	The PAD is a temporary storage area which is always at a fixed offset from HERE. Thus the PAD will move whenever new words are added to the dictionary.
ALLOT	(n ---)	Move the DP along by the value on the stack.
,	(n ---)	Store the 16bit value at the next free space in the dictionary and increment the DP by two.
C,	(b ---)	Store the 8bit value at the next free space in the dictionary and increment the DP by one.
FORTH	(---)	Selects FORTH as the CONTEXT vocabulary. All new word definitions will then become part of the FORTH vocabulary.
DEFINITIONS	(---)	Sets CURRENT to CONTEXT.
VOCABULARY xxx	(---)	Create a new vocabulary called xxx into which new definitions can be defined. Eg. VOCABULARY DATABASE IMMEDIATE This creates a new vocabulary called DATABASE . New words will be added to the DATABASE vocabulary by virtue of the next statement:- DATABASE DEFINITIONS It is best to make the different VOCABULARYs stem from FORTH rather then from other VOCABULARYs. Remember also to set CURRENT=CONTEXT before FORGETing when using VOCABULARYs
CONTEXT	(---)	A User Variable containing the address of the vocabulary which will be searched first.

CURRENT	(---)	A User Variable which contains address of the vocabulary into which new definitions will be compiled.
FENCE	(--- addr)	A User Variable which holds an address beyond which FORGET will not work. The FENCE may be changed to point to words in the users dictionary. Simply find the words NFA and store it in FENCE eg. ' TEST NFA FENCE !
FORGET xxx	(---)	Forgets all words after xxx. It will not proceed beyond the current FENCE address. When using different VOCABULARYs remember to set CURRENT=CONTEXT before issuing FORGET
COLD	(---)	Perform a cold start resetting all User variables to their initial values.
SET-FORTH	(---)	This word is used to reset the dictionary to its cold start value without effecting the source code currently within the VIC.
' xxx	(--- addr)	Searches through the dictionary for the word xxx and leaves its PFA on the stack. If the word is not found an error message is printed. When ' is used within a colon definition the PFA is compiled as a literal into the dictionary.
-FIND xxx	(--- addr tf) found (--- ff) not found	The word xxx is moved to HERE and a search is made first of the CONTEXT vocabualry and then the CURRENT vocabulary. If the word is found a true flag and the PFA address of the word is left. If the word is not found a false flag is left.
LFA	(--- addr)	Converts the PFA into the link-field-address. This points back to the NFA of the previous word in the dictionary. If the word is the last one in the dictionary it has an LFA of Ø.

PFA	(--- addr)	Converts the NFA into the parameter-field-address. This is the address which is used for storage in variables. The PFA is 5 bytes after the CFA .
CFA	(--- addr)	Converts the PFA into the code-field-address. The CFA is the first byte of machine code which is executed when a word is called.
NFA	(--- addr)	Converts the PFA into the name-field-address. The NFA points to the length of the word bit 7 and maybe bit 6 will be set. The actual name of the word then follows. If the word is greater than 3 characters in length the third byte of the name is the exclusive OR of all the characters in the name.
LATEST	(--- addr)	Leave the NFA of the most recently defined word in the CURRENT vocabulary.
TRAVERSE	(addr1 n --- addr2)	Move across the NFA of a word from addr1 leaving either address of the length byte if n=-1 or the address of the last character in the name when n=1. Addr2 is thus the other end of the name from addr1.
ORIGIN	(--- addr)	Leave the address of the first byte of code in the FORTH ROM. ORIGIN actually points to some system default values which are initialised when a COLD start is made.
+ORIGIN	(n --- addr)	Leave the address of the n'th byte relative to the start of the FORTH ROM.

Table 8 - Compiler directives

EXECUTE (addr ---)
When text is being interpreted, the word's CFA is looked up in the dictionary and then EXECUTE is called which actually executes the machine code in the word. EXECUTE can be used to call VIC ROM routines for example:-
HEX FD22 EXECUTE
This performs a cold start to the VIC just as if it has been switched off and then on again.

ENCLOSE (addr1 c --- addr1 n1 n2 n3)
This is the text scanning which is used by WORD to find delimiters between words of source code. Given the address where the search is to begin and the character to search for, the following information is returned. The original address; the byte offset to the first non-delimiter n1; the byte offset to the first delimiter after the text n2 and the byte offset to the next non-delimiter. An ASCII null is treated as an unconditional delimiter and ENCLOSE will not proceed beyond it.

ERROR (n ---)
Prints the word which caused the error and error message n. See table of error messages for more details.

?ERROR (t/f ---)
If the flag is true message 1 is printed.
Error message number: 1.

?EXEC (---)
Issues an error message is FORTH is not interpreting.
Error message number: 5.

?COMP (---)
Issues an error message if FORTH is not compiling.
Error message number: 4.

?STACK (---)
Issues an error message is the stack is full, ie less than STACK-BOTTOM or if the stack is empty.
Error message number: 2 if empty.
Error message number: 3 if full.

?PAIRS (n1 n2 ---)
Compares the two values on the stack and issues an error message if they are not the same. Used in syntax checking for structure words.
Error message number: 6.

?LOADING (---)
Issues an error message if FORTH is not within the LOAD command.
Error message number: 8.

?CSP (---)
Issues an error message if the stack position differs from the value saved in CSP .
Error message number: 7.

>IN (--- addr)
A user variable which holds the current position of the compiler within the source code. When text is being taken for interpretation it is moved to the TIB where >IN can be indexed through it. When source code is being compiled via the LOAD command >IN indexes via the SCREEN address.

!CSP (---)
Save the current stack position into the User variable CSP .

COMPILE (---)
When COMPILE is used within a colon definition the next word in the source code will be compiled into the dictionary. See the CASE example program for further details.

[COMPILE] (---)
This will force the compilation of an immediate word which would otherwise have executed within a colon definition. See the CASE example program for further details. The word is IMMEDIATE.

[(---)
Suspend compilation within a colon definition. Any following words are interpreted rather than compiled into the dictionary. This allows calculations or compilation execeptions to be performed before continuing with compilation. The word is IMMEDIATE.

] (---)
Start compiling to the end of the colon definition or until [terminates execution.

WORD (c ---)
Read characters from the input stream until the delimiter c is found. The text is moved to HERE as it is taken from the input stream. The character count is held in the first byte. Leading occurences of c are ignored. If BLK is non-zero text is read from the appropriate screen otherwise it is taken from the TIB .

CREATE xxx
The next word in the input stream creates a new dictionary entry called xxx . Only the NFA and the LFA are formed and the name is smudged.

SMUDGE (---)
Modifies the NFA of the LATEST word in the dictionary making it available for general access using the FIND command. If SMUDGE is issued again the word becomes unavailable.

: xxx (---)
Begin compiling. The word xxx is created in the CURRENT vocabulary and error checking information is stored for later use. The word is IMMEDIATE .

; (---)
Terminate the definition making the word available for use if an error condition does not exist. The word is IMMEDIATE .

;S (---)
Stop compiling a screen and return control to the user.

STATE (--- addr)
A User variable which tells FORTH if it is interpreting ($ØØ) or compiling ($CØ).

INTERPRET (---)
The interpreter which will sequentially execute or compile words into the dictionary depending on the value in STATE . If the word cannot be found in the dictionary an attempt it made to convert it into a number according to the BASE , if this fails an error message is given.

IMMEDIATE (---)
Makes the LATEST word execute when used within a colon definition. All the structure words are IMMEDIATE because they have to compile more code than simply the CFA of the word. See the CASE example for more details.

ABORT (---)
Stop execution and perform an abort which sets all the stacks to their cold start values and then returns control to the user on Screen Ø. The source code and code compiled into the dictionary is unaffected.

QUIT (---)
Stop execution, clear the return stack and return control to the user on Screen Ø. The source code and any code compiled into the dictionary is unaffected.

LATEST (--- addr)
Leave the NFA of the last word in the CURRENT vocabulary.

LITERAL (n ---)
If compiling then compile the 16bit value on the stack into the dictionary. The definition is immediate so it will execute during a colon definition.

Table 9 - Input/Output

Word	Stack	Effect
KEY	(--- n)	Wait until a key is pressed at the keyboard or a non-zero value is received from the IEEE.
ØKEY	(--- n)	Accept any key, it will return immediately if no key is being pressed. This is the preferred method for accessing the IEEE.
EMIT	(n ---)	Print the byte n out on the selected output device.
CR	(---)	Transmit ASCII 13 to the selected output device. Carriage return.
SPACE	(---)	Transmit ASCII 32 to the selected output device. Space.
SPACES	(n ---)	Send n spaces to the output device.
BL	(--- 32)	Push ASCII space to the stack.
CLR	(---)	Clear the video screen.
HOME	(---)	Home the VIC's output cursor.
P	(x y ---)	Position the VIC's output cursor at the x,y coordinate on the screen. Ø,Ø is at the top left hand corner.
EXPECT	(add n ---)	Wait for characters from the input device and save the text to the address. Stop receiving characters when either the count n has been reached or a carriage return is received. EXPECT will accept any number of characters from the input device. Note that ASCII zero is ignored. EXPECT stores 2 zeros after the last character received.

TEXT	(---)	TEXT is the preferred word for input from the keyboard as it traps all the cursor control keys apart from delete and a left pointing arrow is put to the screen to show the next entry position. Only carriage return will escape from TEXT . TEXT is moved to the PAD address. The number of bytes input is held in the first byte.
#IN	(--- n1)	A special form of TEXT which will only accept numbers from the keyboard as well as a decimal point. A leading minus sign may be input in the first position of the number but is ignored at all other places. If a decimal point is input its position is returned in DPL.
TYPE	(addr n ---)	Transmit n bytes from addr to the selected ouptut device.
-TRAILING	(addr n1 --- addr n2)	Adjusts the character count n1 to remove any trailing blanks. Used in the form:- PAD 30 -TRAILING TYPE The value 30 will be adjusted as necessary.
."	(---)	When used in a colon definition the text up to the next " is compiled into the dictionary. When the word later executes the text is sent to the selected output device. When used outside a colon definition the text between the quotes is printed immediately. The maximum length for a text string is 127 bytes. (dot-quote)
?TERMINAL	(--- t/f)	Leaves true or false depending on wether the STOP key is being pressed.
TIB	(--- addr)	User Variable pointing to the area where words will be interpreted. Set >IN to 0 before executing the command and TIB must be reset to its original value after use.

Table 10 - IEEE

```
OPEN      ( lf dn sa addr count --- )
opens a file using IEEE syntax.
lf   - logical file number.  This will be the value used for all
       future communications with the file. Range 1-255. It is
       a good practice to name the lf after either the device
       number or the secondary address.
dn   - device number. The following are the normal devices used
       on the IEEE.
     0 - keyboard
     1 - cassette
     2 - RS232
     3 - screen
     4 - printer
     8 - disk
sa   - secondary address. This communicates special information to
       the peripheral when the file is opened. The sa of interest are
     0 - open a cassette file for reading.
     0 - open the disk directory file for reading.
     1 - open a cassette file for writing.
     2 - open a cassette file for writing and send EOF (end-of-file)
         when the file is closed.
    15 - open the disk command/error channel.
addr - the address of the filename
count the number of characters in the filename.
If no filename is required set the count to either 0 or 255.

CLOSE     ( n --- )
close logical file n.

IN        ( n --- )
set logical file n for input.

OUT       ( n --- )
set logical file n for output.

OFF       ( --- )
restore  normal  I/O  devices  (these  being  keyboard  and  screen
respectively)

STATUS    ( --- n )
Fetch the current device status.
  bit no.       meaning
    0        Time out on write
    1        Time out on read
    2        Short block (cassette only)
    3        Long block (cassette only)
    4        Unrecoverable read error (cassette only)
    5        Checksum error (cassette only)
    6        EOI (end of file)
    7        Device not present
```

Source code load and save commands

SLOAD xxx (---)
Loads the file named xxx into storage RAM, appending screens onto those already in RAM. The filename is optional. If a disk system is being used it is best to work with SEQential files. Eg.

SLOAD filel

If possible do not abort a load attempt as this can cause corruption to the text pointer TP . If you do abort a SLOAD it is best to use the word RESET , which resets TP only and try again. If there is not enough RAM in the VIC to enable the whole of the file to be loaded an 'out of memory' message will be given and FORTH will reset the TP back to its position before the load was started.
Beware of loading non-existant file names, if this happens type:-
1 TP +!
Do not abort part way through a load as unpredictable corruptions occur.
See COPY for information on duplicate screens.

SSAVE xxx (---)
Saves all the screens currently in RAM to the number selected in DEVICE . If the disk system is being used it is best to work with SEQuential files as when it comes to load them again no control codes are required. Eg.

```
SSAVE filel          ( cassette operation )
SSAVE Ø:filel,s,w    ( disk operation )
```

A file name is not required for cassettes but the disk will indicate an error condition.

DEVICE (--- addr)
This is a User variable which contains the current storage device. When the system is switched on device is set to 1 for cassette operation. When device is changed to 8 the source code will be loaded/saved to disk.

Table 11 - Error messages

The following error messages will be printed when certain error conditions exist within FORTH. The word that caused the error message will be printed followed by the message itself. Unless otherwise stated QUIT is performed after the error message has been printed.

SAY WHAT?
The word can not be found in the dictionary and it can not be converted into a number according to the current numeric base. Eg.
JHKH

STACK FULL
The parameter stack can accept no more values as the value in STACK-BOTTOM has been exceeded. Eg.
: TEST SP! 100 STACK-BOTTOM ! 1000 0 DO I ?STACK LOOP ;

STACK EMPTY
An attempt has been made to remove a value from the stack when it is empty of values. Eg.
SP! .

ISN'T UNIQUE
The word being defined already exists in the dictionary. Compilation will continue. Eg.
: FORTH ;

COMPILATION ONLY
An attempt is being made to use a word which only works when used within a colon definition when the compiler is working. Eg.
IF

EXECUTION ONLY
An attempt is being made to use a word which can only be executed outside a colon definition when the compiler is not working. Eg.
: TEST : ;

CONDITIONALS NOT PAIRED
It is not possible in FORTH to have unbalanced structures. This message will be printed when FORTH detects this error. Eg.
: TEST BEGIN IF AGAIN ;

DEFINITION NOT FINISHED
The definition has not been finished because the current stack position does not match that saved in CSP when the word was started. Eg.
: TEST IF ." TRUE" ELSE ." FALSE" ;

USE WHEN LOADING
The word being used only has meaning when screens are being compiled. Eg.
-->

BRANCH TOO FAR
The 6502 instruction 'BEQ' - branch when equal to zero - is used extensivly within this version of FORTH for branching when conditions are false. For example:-

```
: TEST  IF  code  THEN
        ------------->
```

IF compiles code which will test the top value of the stack, if it is false branch to the code following THEN . A relative branch using BEQ is actually compiled into the dictionary for the sake of speed. IF thus compiles 5 bytes of code.

```
JSR ZBRAN     ;Set the 6502 status register zero flag if value zero.
BEQ ATHEN     ;if the flag is set branch to the code following then.
```

The instruction only has a range of 127 bytes or approximately 40 word definitions and only poor programming will bring it to light. If the error is given, simply move some of the code to another word definition; the loss of speed will be negligible. Another word uses BEQ as well, this being the string printing command ." (dot-quote), this also has a limitation of 127 bytes between quotes. Eg.

```
: TEST ." This is a very long string of characters which will be used
to generate the error message. The distance between quotes is limited
to 127 characters" ;
```

IS PROTECTED!!
An attempt has been made to forget a word that is in the protected dictionary when using the word FORGET . The word FENCE can be altered to change the point at which FORGET works. Eg.

```
FORGET FORTH
```

Table 12 - Numeric Formatting

Word	Stack	Effect
<#	(---)	Begin conversion.
#	(ud --- ud)	Convert one character using the current numeric base and add it to the output string. If there are no characters left a zero is produced.
#S	(ud --- 0 0)	Convert all digits in the unsigned double number leaving 0 0 on the stack.
HOLD	(c ---)	Used between <# and #> to insert the ASCII character c into pictured numeric conversion. Eg. 43 HOLD This will position a plus sign at the current position in the conversion.
SIGN	(n ud --- ud)	Add a hyphen to the output string if n is negative. ud is unaffected.
#>	(ud --- adr n)	End conversion. Leaving the address of the string and its length.
HLD	(--- addr)	A User variable which holds the address of the latest character of text during numeric output conversion. Note that it moves from the start of the PAD and moves backwards.

Table 13 - Defining Words

CONSTANT xxx (n ---)
xxx (--- n)
Creates a constant xxx with the value n. When xxx is later executed n is placed onto the stack.

VARIABLE xxx (---)
xxx (--- addr)
Creates a variable called xxx. When xxx is later executed the PFA or storage address is returned on the top of the stack.

<BUILDS xxx (---)
When used within a colon definition <BUILDS will create a new word called xxx with a dummy execution word. It will be altered later to point to the code following DOES> .

DOES> (---)
Modifies the CFA of the word being created to point to the code following DOES> in such a way as to leave the PFA of the word on the stack before the code starts to execute.

: xxx (---)
Create a smudged header in the dictionary and store the current stack position in CSP .

CREATE xxx (---)
Position the name xxx at the next dictionary space, leave it smudged so that it will not be found until more work has been done on it. All defining words use CREATE .

Table 14 - Predefined Control Structures

```
IF  xxx                          (flag --- )
        ELSE  yyy  THEN
```

IF tests the flag on the stack and when true (non-zero) executes xxx and continues after THEN . When the flag is false (zero) yyy is executed and continues after then.

```
DO  xxx                          (limit index --- )
       LOOP
```

DO takes the limit and index and transfers them to the return stack. xxx is executed until the index=limit and then continues after LOOP. Note that xxx is not executed when index=limit.

```
DO  xxx                          (limit index --- )
      +LOOP                      (incr --- )
```

This is the same as DO ... LOOP except +LOOP expects a value on the stack as an increment.

```
LEAVE                            ( --- )
```
Leaves the DO ... LOOP at the next LOOP or +LOOP.

```
BEGIN xxx
         UNTIL                   (flag --- )
```

Execute xxx until the flag tested by UNTIL is true (non-zero)

```
BEGIN xxx
         WHILE                   (flag --- )
              yyy REPEAT
```

Execute xxx which will leave a flag to be tested by WHILE . If the flag is true yyy is executed and FORTH jumps back to execute xxx again. If the flag is false execution continues after REPEAT.

```
BEGIN xxx AGAIN                  ( --- )
```

Continually execute xxx and leave only if QUIT or ABORT are encountered.

The following words can all be used to fetch LOOP index values for use between DO ... LOOP

R@ (--- nl)
Copy the top of the return stack.

I (--- nl)
Copy the top of the return stack. Used to fetch the loop index. Same as R@.
DO I . LOOP

J (--- nl)
Copy the third item of the return stack to the parameter stack. Used to fetch a second loops index.
DO ... DO J . LOOP LOOP

K (--- nl)
Copy the fifth item of the return stack to the parameter stack. Used to fetch a third loops index.
DO ... DO ... DO K . LOOP LOOP LOOP

Table 15 - Special VIC and 64 Functions

SET	(n --- nl)	Accept a number between Ø-7 and leave nl which corresponds to the bit number being set.

n	nl
Ø	128
1	64
2	32
3	16
4	8
5	4
6	2
7	1

RESTORE	(n ---)	Sets the NMI function. When FORTH is powered on the restore key has no effect. To enable programs to be be debugged effectively the restore key can be switched on. If n is 1 FORTH will perform ABORT whenever the restore key is pressed. When n is 2 FORTH performs the SET-FORTH operation followed by ABORT . It is up to the user to decide which one to use. The first option will recover from most crashes except those which have corrupted the dictionary. Hence the 2nd option.
CONTROL	(--- addr)	A User variable which contains the address of the code which will execute when the key F1 is pressed. See text on editor for information on applying your own functions to F1.

The Following Functions are only available on the VIC-FORTH cartridges.

SOUND	(n reg ---)	Set the sound register 1-4 with the value n. n lies in the range 128-252. If n is less than 128 no sound is produced. Register 4 is white noise.
VOLUME	(n ---)	Sets the volume of the sound being produced. n is in the range Ø-15 where Ø is no volume.
COLOUR	(b f ---)	Set the background and foreground colours of the VIC screen.

EXAMPLE FORTH PROGRAMS

The following programs are intended as being both instructional and useful. Don't just read through them and then type them into your computer - actually work out what is happening and why the code is there, for that is the only way in which you will learn how to program in FORTH.

```
SCREEN NO: 5
------------------------------------------------------------
( SOME EXAMPLE FORTH WORDS)
: EG1  25 2 4 + / .   ;       : EG2  + / . ;
: EG3  17 4 /MOD . . ;        : EG4  9 4 5  */ ;

: EG5  ." This is some text" cr ;
: EG6  TEXT PAD COUNT TYPE ;
: EG7  #IN . ;


: EG8   100 0 DO R@ . LOOP ;  : EG9 23 0 DO R@ . 3 +LOOP ;
: EG10  0 100 DO R@ . -1 +LOOP ;

: EG11  IF ." The number was 11"
        ELSE ." The number was not 11"  THEN ;

: EG12  BEGIN KEY 89 = UNTIL ;
------------------------------------------------------------
```

Some example FORTH words

EG1 When numbers are included within a colon definition they are converted into literals. When the literal is encountered as the word executes the number is moved directly to the stack. Because the numbers have been converted to 16bit values when the word EG1 was compiled, literals will work far faster in FORTH than a similar BASIC program, which has to go through the time consuming process of converting the number into a more useful internal form (usually binary floating point) each time the number is encountered.

In this example three numbers are moved to the stack, the top two values (2 4) are then added and the result (6) is left on the top of the stack. This top value (6) is then divided into the number below it (25), leaving the result on the stack (4). Because the maths is integer the number is rounded down.

```
EG1 4 ok
    ----
```

EG2 This behaves in the same way as EG1 except it requires 3 numbers on the stack before the word is called.

```
45 7 2 EG2 5 ok
           ----
```

EG3 Quite often it is necessary to divide two numbers and to know how large the remainder is. The word MOD will divide the top value on the stack into the one below it and leave the remainder on the stack (eg. 9 4 MOD . 1 ok). The example however is /MOD which leave both the quotient and the remainder on the stack.

```
EG3 4 1 ok
    ------
```

EG4 This example combines a multiply and a divide into one word called star-slash it is especially useful because a 32bit value is maintained between the multiply and divide, resulting in more accuracy when large numbers are being used.

```
EG4 7 ok
    ----
```

EG5 This is an example of how text can be included in a FORTH word. All the characters between the quotes are compiled into the dictionary. Note that the quote at the end of the string does not have to be seperated by spaces as it is not a FORTH word. The word ." (dot-quote) is simply scanning the text ahead to find a " (quote). A string is limited to 127 characters in length.

```
EG5 This is some text ok
    --------------------
```

EG6 This word allows the user to enter a string of up to 64 character to the PAD, the temporary storage area. The number of characters entered is held at the first byte of the PAD. The word COUNT fetches the byte at the address on the stack and leaves this along with the original address plus one. Thus it is especially useful for string handling.

```
EG6 Hello FORTH  Hello FORTH ok
                 --------------
```

EG7 Uses a special form of text input which allows only numbers to be entered. The number is converted to a 16bit value and put onto the stack.

```
EG7 5 5 ok
     ----
```

EG8 This is a simple incrementing loop which uses special code to make it faster then +LOOP. The return stack is used to hold numbers which would otherwise interfere with the normal parameter stack. Values can be moved to and from the return stack using the words >R (to-R), R> (R-from) and R@ (R-fetch). When loops are being used the words I , J , and K can be used to access the current value of the loop. In this example the numbers Ø to 99 will be printed on the screen. LOOP stops when the limit is reached so the last value will not be printed.

```
EG8 Ø 1 2 3 4 5 6 7 .... 94 95 96 97 98 99 ok
    ----------------------------------------
```

EG9 This example is similar to the above except it used the word +LOOP instead of LOOP. +LOOP requires a value on the stack to tell it by how much it is to increment (or decrement) the value on the return stack.

```
EG9 Ø 3 6 9 12 15 18 21 ok
    ---------------------
```

EG1Ø This is a decrementing loop.

```
EG1Ø 1ØØ 99 98 97 96 .... 3 2 1 ok
     ----------------------------
```

EG11 This is the main decision making command available in FORTH from which all others can be derived. IF expects a value on the stack, when the value is zero it is false. When the value on the stack is non-zero it is true.

There are two types of IF statement, the simple IF :-

IF true THEN continue from here

If the value on the stack is true the code between IF and THEN is executed. If the value on the stack is false the code skips directly to the code following THEN. The other type of IF statement is:-

IF true part ELSE false part THEN continue from here

This is the same as the first type of IF statement except that when a value is false the code between ELSE and THEN is executed.

```
12 EG11  The number was not 11 ok
         ------------------------
11 EG11  The number was 11 ok
         --------------------
```

EG12 This is one of the repeating structures (others are BEGIN...AGAIN and BEGIN...WHILE...REPEAT). The program will continually execute the code between BEGIN and UNTIL until the value on the stack preceeding UNTIL is true; the code 89 = compares the value of the key pressed by the user against 89 (ASCII for y), it will leave 1 if it is the same. Control will only return to the user when this key has been pressed.

```
EG12 ok
     --
```

```
SCREEN NO: 80
-------------------------------------------------------------
: LISTER  16 0 DO   SCREEN I 64 * +    64 TYPE  CR   LOOP ;

: LIST    BLK @ SWAP EXPAND   LISTER   EXPAND ;

: LINE    64 0 DO   45 EMIT   LOOP CR ;

: DUMP    4 4 0 0 0 OPEN  4 OUT
            CR CR   DUP   ." SCREEN NO: " . CR
            LINE  LIST  LINE  CR CR CR
          OFF   4 CLOSE ;
```

Listing a screen to a printer

If you have a printer attached to your VIC it is obviously useful to have a permanent record of programs. The above program shows how this can be achieved easily.

The program is also a good example of how to access screens which have been compressed. Here is what each word does:-

LISTER There are 1024 characters to a screen organised as 16 rows of 64 characters. It is obviously necessary to send this to the printer in this format or it will look a mess. So after every 64 characters a carriage return is sent CR and a new line started. The code SCREEN I 64 * + gives an address relative to the base address of the screen storage area.

LIST This expects a screen number on the stack when it is called. The current screen is saved to the stack and the desired screen is expanded from storage into the FORTH screen RAM. LISTER prints out the screen, then EXPAND is called again to bring back the original screen using the value which was left on the stack.

LINE Simply prints a line of 64 dashes.

DUMP This is the main calling word. It opens a file to the printer, which is device 4 and sets it for output. The screen number is printed and then the screen itself bounded by lines. Finally the bus is switched off and the file closed.

Notice that I have followed the simple rule explained in the IEEE section. The program can quite easily be tested without having to open the file to the printer. The routines LISTER , LIST and LINE all work equally well without the file having to be open, thus speeding development time.

```
SCREEN NO: 1
------------------------------------------------------------
: DRIVEØ ." $Ø" ;
: OPENDIR  1 CLOSE   1 8 Ø ' DRIVEØ COUNT OPEN  1 IN ;

: DROPTWO  ØKEY ØKEY DROP DROP ;
: NUMBERS  DROPTWO   ØKEY ØKEY 256 * + ;

: READ   BEGIN  ØKEY DUP EMIT  Ø= STATUS OR  UNTIL ;

: DIRECTORY  NUMBERS STATUS Ø= IF . THEN READ CR ;

: $Ø  147 EMIT OPENDIR DROPTWO  BEGIN DIRECTORY STATUS
      ?TERMINAL OR UNTIL   1 CLOSE OFF
      ." press a key to continue"  KEY DROP 147 EMIT
```

Reading the disk directory

The useful example will read the directory from the disk drive.

DRIVEØ sets the filename, which for reading the directory is $Ø (if the command $Ø:FORTH* was given as the filename then only those files on the disk with the word FORTH in the first five characters would be displayed.

OPENDIR ensure that logical file 1 is closed and then open the directory. Notice how FORTH searches the dictionary for the compiled word DRIVEØ using the ' word, this will leave the PFA on the stack. By chance this address points to the length of the text which follows thus giving OPEN all the parameters it requires to open the file.

DROPTWO reads in two characters and drops them immediatly.
NUMBERS drops the first two characters it receives and then converts the next two characters to a 16bit number.
The reason for this manipulation of the directory is that it is sent from the disk to the computer as a BASIC program so that the BASIC LIST command will work. A BASIC line looks like this:-

llnnxxxxxxxxxxxxxxxxxxx0llnnxxxxxxxxxxxxxxxxxx0llnnxxxxxxx000

ll is the link address pointing to the next line.
nn is the line number.
Ø (null) is the end of link mark.

READ accepts text and displays is on the screen until either the STATUS word changes or a null is received.

DIRECTORY This will fetch the line number, display it if the STATUS is zero and then read in the rest of the line.

$Ø is the main word. The screen is cleared and then the directory is displayed until either the stop key is pressed or the end of the directory is reached (as indicated by a change in STATUS). The message is then displayed and FORTH will wait until a key is pressed.

```
SCREEN NO: 81
-------------------------------------------------------------
: CASE     ?COMP  CSP @ !CSP 4 ; IMMEDIATE

: OF       4 ?PAIRS  COMPILE OVER  COMPILE =   [COMPILE] IF
                     COMPILE DROP  5 ; IMMEDIATE

: ENDOF    5 ?PAIRS     [COMPILE] ELSE   4 ; IMMEDIATE

: ENDCASE  4 ?PAIRS   COMPILE DROP
     BEGIN  SP@ CSP @ = Ø=  WHILE   [COMPILE] THEN   REPEAT
                 CSP ! ; IMMEDIATE
```

The Case Structure

Pascal is a language which has been given much publicity because of its structured approach to programming. FORTH has all the structural control of Pascal, except it lacks the Case structure, so here it is!

The case structure being given here was designed by Dr. Charles E. Eaker and was published in the excellent 'FORTH Dimensions' magazine from the FORTH Interest Group.

Notice that all the words have been made IMMEDIATE . This has the effect that when the compiler comes across them it will execute them rather than compile their CFA address into the dictionary.

CASE makes sure that it is in a colon definition using the word ?COMP. The current stack position is pushed to the stack and will be restored by ENDCASE . The User variable CSP is set to the current position of the stack pointer and will be used later to resolve forward references. Finally a 4 is pushed to the stack to help with syntax checking. CASE compiles no code into the dictionary.

OF first checks to see that it has been preceded by either a CASE or an ENDOF by using the comparison check ?PAIRS . If the syntax is in order then some code is compiled into the dictionary. The word COMPILE takes the next word in the source code and compiles a jump to subroutine (JSR) to its CFA . The structure word IF is compiled into the dictionary using the word [COMPILE] which is used to compile immediate words. IF leaves a 2 on the stack for syntax checks later, a dummy branch is compiled into the dictionary. Two other numbers are left on the stack which will be used later to resolve forward branches. OF leaves 5 on the stack for syntax checking.

When OF has finished compiling the following code will have been added to the dictionary:-

```
OVER = IF DROP
```

ENDOF first checks to see that it was preceded by OF . It then compiles the IMMEDIATE word ELSE into the dictionary using [COMPILE] . When ELSE executes, the branch ahead left by IF is resolved. Another branch ahead is needed by ELSE , dummy values are left for now and ENDCASE puts in true jump addresses later. A 4 is left on the stack for either OF or ENDCASE to use as syntax checks.

ENDCASE makes sure that it was preceded by either CASE or ENDOF . The code for DROP is compiled into the dictionary and all unresolved branches are sorted out by compiling THEN into the dictionary. Since there may be any number of them ENDCASE checks the current stack position against the value stored in CSP and will continue to resolve branches until CSP equals the stack pointer. When FORTH has finished this it restores the stack position back to the value when CASE was first started.

Here is an example for using CASE :-

```
: TEST  CASE
           Ø OF  ." zero "    ENDOF
           1 OF  ." one "     ENDOF
           2 OF  ." two "     ENDOF
           ." What was that? "
        ENDCASE  ." Try another" ;

Ø TEST zero Try another ok
      ------------------

5 TEST What was that? Try another
      -------------------------
```

The above is obvious in meaning, try the following:-

```
: TRIAL  Ø OVER =
           IF  DROP ." zero"
            ELSE  1 OVER =
              IF DROP ." one"
               ELSE 2 OVER =
                 IF DROP ." two"
                   ELSE ." What was that?"
                 THEN
              THEN
           THEN  ." Try another" ;
```

It is exactly the same! CASE generates exactly the same code as that used by TRIAL but in a far more readable form for us humans.

```
CASE     ( --- csp 4 )

OF       ( 4 --- addr1 Ø 2 )

ENDOF    ( addr1 type1 2 --- addr2 1 4 )

ENDCASE  ( csp  addr2 1 4 --- )
```

OF and ENDOF may leave multiple values on the stack which ENDCASE will resolve.

```
SCREEN NO: 2
------------------------------------------------------------------
FORGET TASK     : TASK ;
VARIABLE LEVEL    VARIABLE SEED

: RND    SEED @ 259 * 3 +    32768 AND    DUP SEED !
         32767 */ ;

: BOMBPOS   22 RND   21 RND 1+    P ." ." ;

: SETBOMBS   5 * Ø DO   BOMBPOS   LOOP ;

: SETUP    Ø 3 SOUND   Ø 2 SOUND    15 VOLUME
           CLR ." Game level (1-1Ø) " #IN CLR
           1Ø MIN   1 MAX   DUP LEVEL !   SETBOMBS ;


-->
------------------------------------------------------------------
```

An arcade type game called MineSweeper

This is an arcade type game which I have called MineSweeper. Here is the scenario:-

You are captain of a small space ship patrolling a quadrant of space. An alien ship is in the process of depositing mines in the quadrant. It is up to you to try and keep the quadrant clear so other space traffic can move through. Unfortunately you do not have the weaponry to destroy the alien ship. You can only ram it while the mine is being laid. When the ship is rammed it drops into hyperspace and will reappear at another place on the screen. While your ship can withstand this sort of treatment it can not withstand many inpacts from the mines which have been dropped already. However there is a period when the alien ship stays in hyperspace for a longer period having thrown 5 extra mines at random into the quadrant. During this short period you can move around with much more freedom and collect mines which have already been laid.

Okay so now you've read the book, write the program....

When developing programs it is a good idea to keep the dictionary as clean as possible hence the use of the statement in the first line. Before loading the program for the first time it is necessary to enter : TASK ; so that it can be forgotten and then recreated again. Once TASK has been defined once it will always be in the dictionary and it means that there will always be a common starting point in the dictionary.

RND generates a random number between Ø and the number on the stack when RND is called. The current random number is used as a seed for the next one.

BOMBPOS chooses a random position on the screen which is greater than line 1 and puts a dot there which represents a mine.

SETBOMBS is called with a value on the stack. This number is multiplied by 5 to give a number of mines depending on the level of the game requested by the user.

SETUP The sound is switched off and the volume is set to maximum for later use. The user is asked to input a game level and FORTH makes sure this is between 1 and 10 using the very useful MIN and MAX words. The game level is set and the number of starting mines dropped onto the screen.

--> This is a FORTH word which tells the compiler to continue compiling with the next screen. The current screen is compressed to RAM, the value in BLK is incremented by one and the screen brought from storage. >IN is reset to zero. This is a far more efficient way of compiling programs than using a multitude of LOAD commands.

```
SCREEN NO: 3
-----------------------------------------------------------------
VARIABLE XPOS          VARIABLE YPOS
VARIABLE DIRECTION

: CHANGE  ØKEY >R Ø
               R@ 157 = IF  6Ø  THEN
               R@  29 = IF  62  THEN
               R@ 145 = IF  94  THEN
               R@  17 = IF 214  THEN
               R> DROP   ?DUP IF  DIRECTION !  DROP  THEN ;

: MOVE XPOS @ YPOS @ P SPACE
     DIRECTION @ 6Ø  =  XPOS @ Ø  >  AND IF  -1 XPOS +!  THEN
     DIRECTION @ 62  =  XPOS @ 21 <  AND IF   1 XPOS +!  THEN
     DIRECTION @ 94  =  YPOS @ 1  >  AND IF  -1 YPOS +!  THEN
     DIRECTION @ 214 =  YPOS @ 21 <  AND IF   1 YPOS +!  THEN
       XPOS @ YPOS @  OVER OVER P  22 * + 4Ø96 + C@
                      DIRECTION @ EMIT ;      -->
-----------------------------------------------------------------
```

This screen gets the players moves and translates them into directions for the ship.

CHANGE scans the keyboard. The word ØKEY is used rather than KEY because it does not wait for the user to press a key. If no key is pressed Ø is pushed to the stack. The word KEY will wait forever until a key is pressed which is not suitable for a real time program.

The value returned by ØKEY is pushed to the return stack where it is easily accessed. Zero is pushed to the stack before the comparisons so a test can be made later to see if a legal key has been pressed. The value on the return stack is compared against the 4 cursor key values.

If you find the cursor keys difficult to use then set your own; the code KEY . will return a value for any key pressed on the keyboard. It would be a fairly simple matter to change the program to work with a joystick. See VIC Programmers Reference Guide (published by Commodore) for details.

The value left by a good comparison is a character code for the symbol which indicates the direction of travel. The return stack is cleaned up by dropping the value which was pushed to it. ?DUP will duplicate the value on the stack if it is non-zero. If it is non-zero then the direction is set to the value on the stack. However there is still the

original zero on the stack, this is dropped. If the value on the stack was zero there is nothing left after the logical test by IF .

MOVE makes the ship move around the screen. The old position of the ship is cleared by taking it out with a space. BEWARE that null strings such as ." " have a very strange effect on FORTH and should not be used. Spaces are produced either by SPACE or n SPACES where n is any value. If the direction and the limit in that direction are in bounds the ship position is changed. The output location is moved to the new ship position and the current value at the location is brought to the stack.

```
SCREEN NO: 4
-------------------------------------------------------------
VARIABLE SHIELDS   VARIABLE SCORE   VARIABLE MAXSCORE

: DELAY   10000 0 DO LOOP ;
: NAME ." VIC                    " ;

: MAX?  SCORE @ MAXSCORE @ > IF ." You have then maximum  score"
    SCORE @ MAXSCORE !    CR ." What is your name?" CR TEXT
    PAD 1+   ' NAME 1+  20 CMOVE  THEN
    ." Maximum score by:-" CR  NAME ;

: DAMAGED   -1 SHEILDS +!    14 0 P
       SHIELDS @ 2 = IF  ." SLIGHT"   DELAY THEN
       SHIELDS @ 1 = IF  ." CRITICAL" DELAY THEN
       SHIELDS @ 0 = IF  CLR ." Ship damaged return tobase" CR
       ." You scored: " SCORE ?  CR  0 VOLUME  MAX?
       5 0 DO DELAY LOOP  QUIT THEN ;      -->
-------------------------------------------------------------
```

NAME sets a string which will be modified whenever the maximum score is beaten.

MAX? test to see if the maximum score has be beaten. If it has, the user is asked to input a name. TEXT will accept up to 64 characters from the keyboard storing them at PAD . The count is stored in the first character of the PAD . Since this is not required PAD 1+ fetches the address of the first character of text. ' NAME will return the PFA of the string. If you type ' NAME C@ . you will be given the length of the string. Under no circumstances should the length of the string be changed, any modifications stand a good chance of crashing FORTH. Thus ' NAME 1+ will return the address of the first byte of the string. 20 bytes are then moved from the PAD to the area of the string. Once again if more bytes than the maximum length of the string are moved FORTH will crash.

DAMAGED is quite simple, a message is printed depending on the current value of SHIELDS and when SHIELDS is zero the program stops. If you want the program to re-run automatically it will be necessary to leave either a value on the stack which can be tested when the word returns control to ONETIME and then to GAME which would be fairly messy, or set a VARIABLE which can be tested by GAME directly.

```
SCREEN NO: 5
------------------------------------------------------------
VARIABLE WAITIME    VARIABLE TIME
VARIABLE XSHIP      VARIABLE YSHIP

: ATTACK   128 250 DO   I 2 SOUND   CHANGE MOVE
           46 = IF   1 SCORE +!   4 Ø P SCORE ?  THEN
           2000 Ø DO LOOP   -1 +LOOP    Ø 2 SOUND ;

: SETSHIP  22 RND  21 RND 1+  OVER OVER P ." *"
           YSHIP !  XSHIP !   WAITIME @ TIME ! ;

: HITALIEN   LEVEL @ SCORE +!   4 Ø P SCORE ?
   -2 WAITIME +!  WAITIME @ 30 < IF  1 SETBOMBS  1 LEVEL +!
      ATTACK   50 WAITIME !  THEN  SETSHIP ;

-->
------------------------------------------------------------
```

ATTACK when the player has hit the alien ship 10 times the program dumps five bombs at random and then drops into attack. The player can then move about the screen running over the bombs which have been left. The sound generator starts at a high tone and decends to a low tone. If the value being left behind by MOVE is 46 (ie. a decimal point) the score is incremented by one.

SETSHIP chooses a random position for the enemy ship to appear on the screen which is not the first line, displays the ship and stores the co-ordinates in XSHIP and YSHIP . The countdown timer TIME is set to the current value of WAITIME .

HITALIEN is called when the player runs over the alien ship this word. The score is incremented according to the current level. WAITIME is decremented giving less time for the player to get to the ship before it disappears again. When WAITIME is less than 30 the game moves to the next level and the word ATTACK is called.

```
SCREEN NO: 6
------------------------------------------------------------
: MOVEALIEN   XSHIP @  YSHIP @  P ." ."  SETSHIP
              WAITIME @ TIME !  ;

: SETGAME   50 WAITIME !    40  TIME  !    Ø  SCORE  !    3  SHIELDS  !
                  11 XPOS !       11 YPOS !  214 DIRECTION !
            SETUP  Ø Ø P ." PTS Ø   DAMAGE NIL" SETSHIP ;

: ONETIME   CHANGE MOVE DUP DUP
      43 =  SWAP  42 =  OR  IF  HITALIEN  THEN
                        46 = IF  DAMAGED   THEN
         TIME @ 128 +   3 SOUND
      -2 TIME +!  TIME @ 12 < IF  XSHIP @ YSHIP @ P ." +" THEN
         TIME @ Ø< IF  MOVEALIEN  THEN ;

: RUN   SETGAME  BEGIN  ONETIME  2000 Ø DO LOOP
                         ?TERMINAL  UNTIL  Ø VOLUME ;
------------------------------------------------------------
```

MOVEALIEN changes the alien ship into a mine and then moves the ship to another position on the screen using SETSHIP . The variable TIME is reset.

SETGAME initialises all the variables, requests a game level and prints the status line at the top of the screen.

ONETIME runs through the game once, selecting HITALIEN if your ship has moved over either '*' or '+'. If the ship has moved over a mine '.' then DAMAGED is called. The sound from register three is modified to indicate that the ship is preparing to move. When the time to move is less than 12 the ship changes shape. When the time goes below zero the ship is moved using MOVEALIEN .

RUN is the calling word for the game which the user types to get the game running. The game will play until either the stop key is pressed or the ship runs into three mines in which case QUIT is called from the word DAMAGED .

This is the Mine-Sweeper program converted to run of the Commodore 64. The only changes which had to be made were concerned with the screen size, the very different sound characteristics and some timing modifications.

```
SCREEN NO: 1
------------------------------------------------------------------
FORGET TASK     : TASK ;
HEX

: SETSOUND    0F D418 C!    09 D405 C!   09 D406 C!

: SHAPE D404 C! ;

: DURATION 100 0 do loop ;

: NOTEON   D400 !  17 SHAPE ;
: NOTEOFF          16 SHAPE ;      DECIMAL

-->
------------------------------------------------------------------

SCREEN NO: 2
------------------------------------------------------------------
VARIABLE LEVEL    VARIABLE SEED

: RND    SEED @ 259 * 3 +    32768 AND    DUP SEED !
         32767 */ ;

: BOMBPOS   40 RND   23 RND 1+    P ." ." ;

: SETBOMBS   5 * 0 DO   BOMBPOS   LOOP ;

: SETUP    SETSOUND
           CLR ." Game level (1-10) " #IN CLR
           10 MIN   1 MAX   DUP LEVEL !   SETBOMBS ;

-->
------------------------------------------------------------------

SCREEN NO: 3
------------------------------------------------------------------
VARIABLE XPOS          VARIABLE YPOS
VARIABLE DIRECTION

: CHANGE   0KEY >R 0
                R@ 157 = IF   60   THEN
                R@  29 = IF   62   THEN
                R@ 145 = IF   94   THEN
                R@  17 = IF 214   THEN
                R> DROP    ?DUP IF   DIRECTION !   DROP   THEN ;

: MOVE XPOS @ YPOS @ P SPACE
     DIRECTION @ 60   =   XPOS @ 0   >   AND IF   -1 XPOS +!   THEN
     DIRECTION @ 62   =   XPOS @ 39 <   AND IF    1 XPOS +!   THEN
     DIRECTION @ 94   =   YPOS @ 1   >   AND IF   -1 YPOS +!   THEN
     DIRECTION @ 214 =   YPOS @ 23 <   AND IF    1 YPOS +!   THEN
       XPOS @ YPOS @   OVER OVER P   40 * + 1024 + C@
                       DIRECTION @ EMIT ;      -->
------------------------------------------------------------------
```

```
SCREEN NO: 4
------------------------------------------------------------------
VARIABLE SHIELDS   VARIABLE SCORE   VARIABLE MAXSCORE

: DELAY   10000 0 DO LOOP ;
: NAME ." CBM64                  " ;

: MAX?  SCORE @ MAXSCORE @ > IF ." You have then maximum  score"
    SCORE @ MAXSCORE !    CR ." What is your name?" CR TEXT
    PAD 1+   ' NAME 1+  20 CMOVE  THEN
    ." Maximum score by:-" CR  NAME ;

: DAMAGED   -1 SHEILDS +!    14 0 P
      SHIELDS @ 2 = IF  ." SLIGHT"   DELAY THEN
      SHIELDS @ 1 = IF  ." CRITICAL" DELAY THEN
      SHIELDS @ 0 = IF  CLR ." Ship damaged return tobase" CR
      ." You scored: " SCORE ?  CR  MAX?
      5 0 DO DELAY LOOP  QUIT THEN ;      -->
------------------------------------------------------------------

SCREEN NO: 5
------------------------------------------------------------------
VARIABLE WAITIME    VARIABLE TIME
VARIABLE XSHIP      VARIABLE YSHIP

: ATTACK   1280 2500 DO   I NOTEON   CHANGE MOVE
           46 = IF   1 SCORE +!   4 0 P SCORE ?  THEN
           500 0 DO LOOP  NOTEOFF  -10 +LOOP ;

: SETSHIP  40 RND  23 RND 1+  OVER OVER P ." *"
           YSHIP !  XSHIP !   WAITIME @ TIME ! ;

: HITALIEN   LEVEL @ SCORE +!   4 0 P SCORE ?
   -2 WAITIME +!  WAITIME @ 50 < IF  1 SETBOMBS  1 LEVEL +!
      ATTACK   70 WAITIME !  THEN  SETSHIP ;


-->
------------------------------------------------------------------

SCREEN NO: 6
------------------------------------------------------------------
: MOVEALIEN   XSHIP @  YSHIP @  P ." ."  SETSHIP
              WAITIME @ TIME !  ;

: SETGAME   70 WAITIME !    40  TIME  !    0  SCORE  !    3  SHIELDS  !
                  20 XPOS !       12 YPOS !  214 DIRECTION !
            SETUP  0 0 P ." PTS 0   DAMAGE NIL" SETSHIP ;

: ONETIME   CHANGE MOVE DUP DUP
      43 =  SWAP  42 =  OR  IF  HITALIEN  THEN
                        46 = IF  DAMAGED   THEN
         TIME @ 128 *   NOTEON
      -2 TIME +!  TIME @ 12 < IF  XSHIP @ YSHIP @ P ." +" THEN
         TIME @ 0< IF  MOVEALIEN  THEN  NOTEOFF ;

: GAME  SETGAME  BEGIN ONETIME  1000 0 DO LOOP -1 TIME +!
        ?TERMINAL UNTIL
------------------------------------------------------------------
```